On Large-Scale System Performance Analysis and Software Characterization

Author(s):
Anghel, Andreea-Simona

Publication Date:
2017

Permanent Link:
https://doi.org/10.3929/ethz-b-000212482 →

DISS. ETH NO. 24524

ON LARGE-SCALE SYSTEM PERFORMANCE ANALYSIS AND SOFTWARE CHARACTERIZATION

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH

(Dr. sc. ETH Zurich)

presented by

ANDREEA-SIMONA ANGHEL

Ing. Sys. Com. Dipl. EPF

born on 19.08.1986

citizen of Romania

accepted on the recommendation of

Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Anton Gunzinger, co-examiner
Dr. Gero Dittmann, co-examiner

2017

A dissertation submitted to
ETH Zurich
for the degree of Doctor of Sciences

DISS. ETH No. 24524
Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Anton Gunzinger, co-examiner
Dr. Gero Dittmann, co-examiner

Examination date: July 26th, 2017.

This work was conducted in the context of the joint ASTRON and IBM DOME project and was funded by the Dutch Ministry of Economische Zaken, and the Province of Drenthe.

To my beloved husband and family

Acknowledgements

Doing a PhD has been a great experience for me during which I had the chance to learn how to conduct high-quality research, how to write good scientific publications, how to collaborate in an effective manner with engineers and researchers worldwide and to meet some extraordinary intelligent people. Here I would like express my appreciation to all of those that helped me during this important stage of my life.

First, I would like to extend my sincere gratitude to Prof. Dr. Lothar Thiele for providing me with the opportunity to pursue my doctoral studies under his guidance and for being very supportive throughout the years. I highly appreciate all the discussions that we had during our PhD sessions, which taught me to ask the right research questions and to effectively address them. Furthermore, I would like to thank Prof. Dr. Anton Gunzinger for accepting to co-advise me and for all the feedback during the different stages of my research. Our discussions taught me to pragmatically address a complex research topic and to not forget about the applicability side of research.

I sincerely thank Dr. Gero Dittmann for helping me survive professionally and personally throughout my PhD. I highly appreciate his openness and all his guidance throughout these tough years. I am indebted to him for his permanent encouragement and support. I have learned a lot from him, from complex topics such as system modeling to how to efficiently write a technical paper and professionally coordinate a research discussion.

Moreover, I would like to express my gratitude to Ronald Luijten for all the support and for always believing in me and to Dr. Ton Engbersen and Dr. Martin Schmatz for allowing me to pursue my graduate studies at IBM Research – Zurich as part of the DOME project. Without them I would have not had the chance to work on this very challenging project.

I would like to express special thanks to my DOME P1 colleagues, Dr. Rik Jongerius and Dr. Giovanni Mariani, for all the hard work that we have done together over the last years. We went through many exhausting long technical discussions, but we have always managed to find a good solution. I am also grateful to have built nice memories from the various social events that we have attended together. Within the same project, I also had the opportunity to

supervise two very good master students, which helped me gain a practical insight into several aspects of compilers and software optimizations. For this and all important lessons that I have learned from our collaboration, I thank Laura Vasilescu and Evelina Dumitrescu.

I owe a lot to my current and former colleagues at IBM Research – Zurich, IBM Deutschland R&D GmbH and IBM Research Yorktown (USA) for all the inspiring discussions and for a wonderful work environment throughout the years: Mitch Gusat, Dr. Patricia Sagmeister, Dr. Jonas Weiss, Dr. German Rodriguez Herrera, Georgios Kathareios, Michael Kauffmann, Dr. Cyriel Minkenberg, Dr. Robert Birke, Dr. Peter Altevogt, Dr. Cedric Lichtenau, Dr. Thomas Pflueger, Dr. Jose Moreira and Dr. Jessica Tseng. Special thanks go also to my former and current office-mates, Dr. Anil Kurmus, Dr. Matthias Neugschwandtner, Nathalie Casati, Dr. Florian Auernhammer, Celestine Duenner, Dr. Wolfgang Denzel and Dr. Milos Stanisavljevic.

Furthermore, I am grateful to Charlotte Bolliger and Anne-Marie Cromack from the IBM publications department for proofreading and correcting my publications. I have learned a lot from their suggestions and corrections. I would also like to thank Jilly Fotheringham and Jens Poulsen from the IBM IS team for all the help during the past years.

I would like to also thank all my DOME colleagues from the Netherlands for all the interesting face-to-face meetings, wonderful social events and for the myriad discussions on innumerable topics, in particular to Albert Jan Boonstra, Bram Veenboer, Leandro Fiorin, Erik Vermij, Chris Broekema, Stefan Wijnholds and Andre Gunst.

I would like to thank my parents Liliana and Ilie, my brother Radu and his family for being very patient and coping with my sometimes difficult behavior. They have been a continuous support for me and I thank them for never allowing me to give up. I am also grateful to Grit Abe, Adela Almasi and Mareike Kuehn for their wonderful friendship and for all the great moments that we have created together throughout the last years.

Most importantly, I am extremely grateful to my husband Bogdan for his love and permanent support over the years. He has always encouraged me and pushed me to finalize this thesis. Without his constant optimism, I would have probably not arrived at the end of this work.

Zurich, 01.05.2017

Abstract

Over the years, many scientific breakthroughs have only been possible thanks to advances in the field of very large high-performance computer systems. To reach the exascale computing era, these systems will need to further increase their size, performance and energy efficiency. Building an exascale system under stringent power and performance constraints will be a very challenging task for any organization. Project planning requires early estimates of the system size, performance and power consumption. To address these challenges, system designers need holistic methodologies to simultaneously analyze multiple system components and performance metrics. Such methodologies should also be fast so that designers can efficiently analyze a wide range of hardware design points.

In this thesis, we devise tools and methods to enable: (1) a qualitative investigation of the performance and power consumption of future large-scale systems, and (2) an efficient exploration of system design points by loading platform-independent software properties into analytic system performance models. We first decouple the software characterization from performance modeling and extract compute and communication properties inherent to applications. Then, we load the hardware-independent software profiles into analytic processor and network models. Such a methodology is useful for system designers at an early design stage to gain insights into system behavior. The main contributions of this thesis are:

- We introduce PISA (Platform-Independent Software Analysis tool), a framework for extracting architecture- and ISA-agnostic software profiles from sequential and parallel workloads at native execution time. We illustrate how our framework can be leveraged to extract application signatures that impact the system performance.

- We analyze if platform-agnostic software profiles can enable analytic modeling of processor performance and power consumption. We provide the first study of the accuracy of using ISA-agnostic application signatures with two analytic processor models. The results show that we can achieve an average accuracy of 34%, while preserving the relative performance trends across workloads. We also show that the analytic power model preserves the relative trends across hardware systems.

- We study how to analytically model the processor branch miss rate, without simulating the the branch prediction mechanism. We start from a state-of-the-art characterization metric of branch predictability, the branch entropy. We identify the first method to reverse engineer the history size of a branch predictor using branch entropy estimates. We outline the limitations of branch entropy and propose a hardware-independent method to derive analytic performance models of branch predictors. We also introduce a new branch predictability metric that is up to 17 percentage points more accurate than branch entropy.

- We propose a method for estimating the node injection bandwidth effectively sustained by a network by taking into account the application's communication pattern and a network specification. We derive analytic bandwidth models for classes of communication patterns (uniform, shift and 2-dimensional nearest-neighbor) and network topologies (fat-trees, tori, full-mesh and 2D HyperX). The proposed models achieve an accuracy of more than 90% in the majority of cases. The validation results also show that the models can reliably be used to perform design-space exploration across topologies.

- We present the first methodology that estimates the performance of large-scale systems using as input platform-independent software profiles with analytic processor and network bandwidth models. For the most scalable implementation of a representative benchmark of graph analytics, the proposed methodology obtains good correlation (0.92) across different hardware systems when comparing the estimates with supercomputer measurements. This result indicates that our methodology could reliably be used to (1) rank systems by performance, and (2) perform early and fast design-space exploration.

- As case studies, we perform design-space exploration of (1) compute nodes for algorithms used by the biggest and most sensitive radio telescope to be built in the upcoming years (the SKA), and (2) compute nodes and network topologies for a representative application of graph analytics (Graph 500). For the radio astronomy case, we also contribute with a set of application-specific ASIC/FPGA power models, platforms currently not supported by our general-purpose system analysis methodology.

Designing new systems requires a good understanding of the properties of the workloads that will run on them. This understanding is typically obtained through software measurements on existing systems. In this case, the measurement tools need to be scalable and reliable to accurately measure the system bottlenecks. A final contribution of the thesis lies in this area. Out-of-the-box MPI software profiling tools do not differentiate between time spent in data transfer via network and time spent in waiting for messages to be processed on other processes (data dependencies). Thus, we propose a high-precision profiling methodology that quantifies not only the time spent in compute and communication, but also the time spent in inter-process data dependencies. This is relevant for system designers as only the time spent in data transfer can be optimized by optimizing the network. We apply our methodology to a representative benchmark of graph analytics when run on a real supercomputer.

Résumé

Au cours des dernières décennies, des nombreuses découvertes scientifiques ont été rendues possibles grâce aux progrès dans le domaine des systèmes informatiques à grande échelle et de haute performance. Pour atteindre l'époque de puissance de calcul exa-échelle, ces systèmes nécessiteront d'augmenter leur taille, performance et efficacité énergétique. La conception d'un exa-système sous des strictes contraintes de performance et d'énergie sera une tâche très complexe. La planification de projet exige des premières estimations de taille, performance et consommation d'énergie. Pour construire une architecture optimale, les concepteurs de système ont besoin d'une méthodologie holistique pour analyser simultanément plusieurs mesures de performance pour plusieurs composants. En plus, ces méthodologies doivent être rapides pour permettre l'analyse d'un grand nombre d'architectures matérielles.

Dans cette thèse, on propose des outils et des méthodes pour rendre possible : (1) l'investigation qualitative de la performance et consommation d'énergie des superordinateurs de la prochaine génération, et (2) une exploration efficace des différentes architectures en combinant des profils de logiciels, indépendants de l'architecture matérielle, avec des modèles mathématiques de performance et énergie. D'abord on sépare les propriétés des logiciels de la modélisation de performance et consommation d'énergie. On extrait des profils de computation et communication intrinsèques aux logiciels. Ensuite on utilise ces profils avec des modèles mathématiques de processeur et de réseau. Une telle méthodologie est utile pour les concepteurs de système. En effet, dans les premières étapes de conception, il est important de comprendre les comportements des systèmes, en fonction des propriétés des logiciels et architectures matérielles. Les contributions principales de cette thèse sont :

- On propose PISA (Platform-Independent Software Analysis), un outil capable de profiler des logiciels de manière indépendante de l'architecture matérielle et de l'architecture du jeu d'instructions. On utilise PISA pour analyser des logiciels séquentiels et parallèles pendant l'exécution du logiciel. On démontre comment utiliser PISA pour extraire des propriétés qui influencent la performance des systèmes.

- On analyse si ces profils des logiciels permettent la modélisation mathématique de

la performance et de l'énergie des processeurs. On présente la première étude de la précision de la combinaison de tels profils avec deux modèles mathématiques de performance des processeurs. Les résultats montrent une précision moyenne de 34%, en gardant les tendances relatives de performance sur l'ensemble des logiciels. On montre aussi que le modèle d'énergie garde les tendances relatives de consommation d'énergie sur l'ensemble des architectures matérielles.

- On étudie comment modéliser mathématiquement le taux de prédiction de branchement dans un processeur sans simuler le mécanisme de branchement matériel. On commence par l'analyse d'une méthode de l'état de l'art, l'entropie de branchement. On propose la première méthode d'ingénierie inverse de l'histoire globale du mécanisme de prédiction de branchement. On montre aussi les limitations de l'entropie de branchement et on propose une approche de dériver des modèles mathématiques pour estimer la performance des prédictions de branchement. Finalement, on propose une nouvelle mesure pour caractériser la performance de prédiction de branchement qui est de 17 points de pourcentage plus précise que les prédictions basées sur l'entropie de branchement.

- On propose une méthode pour estimer la bande passante d'injection d'un nœud effectivement soutenue par le réseau en fonction du schéma de communication du logiciel et la spécification matérielle du réseau. On présente des modèles mathématiques pour trois catégories de schémas de communication représentatives en matière de calcul de haute performance (uniforme, shift, 2-dimensionnel nearest-neighbor) et quatre topologies de réseaux (fat-tree, tori, full-mesh et 2D HyperX). Les modèles de bande passante ont une précision de plus de 90% dans la plupart des cas. Ils peuvent aussi être utilisés de manière fiable pour analyser rapidement un nombre élevé d'architectures de réseau.

- On présente la première méthodologie pour estimer la performance des systèmes à grande échelle en combinant des profils des logiciels indépendants de l'architecture matérielle et de l'architecture du jeu d'instructions avec des modèles mathématiques des processeurs et réseaux. Pour la plus évolutive implémentation d'un logiciel représentatif pour l'analyse des graphes (Graph 500), notre méthode obtient des bonnes corrélations (0.92) sur l'ensemble des architectures quand on compare avec des mesures des superordinateurs réels. Ce résultat indique que notre méthode peut être utilisée de manière fiable pour (1) classer les systèmes en fonction de leur performance, et (2) analyser rapidement un grand nombre d'architectures dans les premières étapes de conception de système.

- Comme études de cas, on présente une analyse 1) d'architectures de processeur pour les algorithmes utilisés dans le contexte de SKA, le plus grand télescope radio du monde qui sera construit dans les prochaines années, et 2) d'architectures de processeur et réseau pour Graph 500. Dans l'étude de radioastronomie, on propose aussi des modèles d'énergie spécifiques aux logiciels de SKA pour les architectures d'ASIC et de FPGA. Ces deux architectures ne sont pas actuellement modelées par la méthodologie générale d'évaluation de la performance des systèmes à grande échelle mentionnée au-dessus.

En général, la conception de nouveaux systèmes demande une bonne compréhension des propriétés des logiciels qui seront exécutés sur ces systèmes. Cette compréhension vient normalement des mesures des logiciels sur des systèmes actuels. Dans ce cas, les outils de profilage doivent être évolutifs et fiables pour mesurer avec précision l'interaction des logiciels avec les architectures matérielles. Une contribution finale de cette thèse est située dans ce domaine. Les outils de profilage actuels pour les logiciels MPI ne sont pas capables de différencier entre le temps passé en transférant les données via le réseau et le temps passé dans les dépendances de données (en attendant que les données soient traitées avant d'être envoyeés dans le réseau). On propose une méthodologie de haute précision qui peut quantifier pas seulement le temps passé dans les parties de calcul et communication d'un processus, mais aussi le temps passé dans les dépendances des données entre les processus. Cette information est importante pour les concepteurs de systèmes, parce que seulement le temps passé dans le transfert via le réseau peut être réduit en optimisant le réseau. On utilise la méthodologie pour analyser Graph 500 lors de son exécution sur un superordinateur actuel.

Contents

5 Analytic Modeling of Network Communication Performance 89

1 Introduction

Innovations through advanced scientific computing have significantly impacted both science and our society. The prediction of the activity in the earth's mantle, the prediction of storms and weather phenomena, economic forecasts, the study of galaxy evolution, cosmology and dark energy, are only a few examples of complex computational problems that would not have been possible to solve without the development of high-performance computers or supercomputers.

We start this chapter with describing the case for building new supercomputers and our approach to support their design. Section 1.2 presents a background discussion on the challenges of workload characterization and system modeling. The overall problem statement and a discussion of the thesis contributions are presented in Section 1.3, which is followed by a more detailed plan of the thesis in Section 1.4.

1.1 The Case of Modeling Supercomputers

During the last decades, each advance in computing capabilities allowed applications to run with larger problem sizes, on larger amounts of data, faster and often at a lower power consumption than before. In less than a decade, the performance of such systems has increased by two orders of magnitude, from 280 TFlops in 2006 to 93 PFlops in 2016 [11]. The energy efficiency also increased from approximately 200 MFlops/W in 2006 to 6 GFlops/W in 2016 [11]. These impressive improvements were possible because of the, e.g., transistor speed and energy scaling by reducing the semiconductor size (Moore's Law), processor clock frequency increase, supply voltage and dynamic power consumption decrease, micro-architectural advances exploiting the transistor density gains and cache architectures emulating fast data transfers to span the gap between processors and main memory [41].

Hardware-software co-design [125], simultaneous optimization of both applications and hardware, played a crucial role in attaining the Petascale computing age. This method was successfully used, e.g., in the case of the first Petascale supercomputer released in 2009, the IBM Roadrunner (BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 GHz / Opteron DC 1.8 GHz, Voltaire Infiniband). This supercomputer was developed for modeling the decay of the US nuclear weapons stockpile, under the auspices of the National Nuclear Security Administration's Advanced Simulation and Computing program. Four years later, in 2013, even though performance-wise the Roadrunner was still a top high-performance computer, it was shut down for operational cost reasons.

In the upcoming decades, supercomputers are expected to significantly increase their size, performance and energy efficiency, reaching this time the exascale computing age. An example of a real case study that will require an exascale system in the next decade is the back-end processing pipeline of the largest and most sensitive radio telescope in the world, the Square Kilometer Array (SKA), which will be built starting in 2018. The main mission of this telescope will be to enable the collection and interpretation of very weak radio signals from the beginning of the Universe, i.e., from more than 13 billion years ago. According to a study performed in 2014, all the SKA antennas will generate data at output rates in the range of petabits (10^{15}) per second, which is more than 100 times the 2014 rate of the global internet traffic. Part of this data will end up in the SKA back-end supercomputing center where the compute requirements of the scientific applications using the data will be in the range of 10 exa-operations/second [78]. On top of these high requirements, the power budget allocated to the system is expected to not be higher than 20 MW.

Building an exascale system under a 20 MW power consumption constraint will be challenging. Since 2013 until June 2016, the highest performance attained by a supercomputer, according to the Top500 list of the world's top supercomputers, stayed constant at 33 PFlops, with a power consumption of 17 MW. In June 2016, a new Petascale system, the Sunway TaihuLight, capable of 93 PFlops at a slightly better power consumption of 15.37 MW claimed the No. 1 rank in the 47th edition of the Top500 list. This achievement is one encouraging step towards exascale. However, looking at the power efficiency of the most powerful supercomputers, as shown in the June 2016 list, the efficiency ranges from approximately 50 MFlops/W to only 6.6 GFlops/W. To reach exascale, engineers and systems architects will need to find solutions to improve the power efficiency by one order of magnitude, to at least 50 GFlops/W.

Studies like [26] and [112] thoroughly present the technical challenges of the new era of computing. As the semiconductor manufacturing industry has reached the physical limits of voltage scaling [66], technology advances have already started to face challenges due to stringent energy budgets. In planning and designing the next-generation supercomputers, system architects must consider multiple trade-offs and dependencies. Many aspects of a large-scale computing system will need to be taken into account: the gate delays in a given process technology node, the micro-architectural and architectural properties of a compute node, the communication network, the storage, the cooling technologies, the software etc. Thus, *a fast holistic methodology* for simultaneously analyzing multiple system performance metrics for a wide range of hardware design points is required for a successful design of the next-generation supercomputers.

Designing new high-performance and more power-efficient systems needs to be supported by a good understanding of the properties of the software that will run on it. When the system design relies on application analysis via measurements on existing systems, the characterization tools need to be scalable to large problem sizes and reliable to accurately measure the system bottlenecks. We tackle this problem by introducing a scalable technique for accurately characterizing the time spent in parallel workloads in compute, communication and inter-process data dependencies.

When the system design relies on analysis of applications in a hardware-independent manner, tools that measure the applications' inherent properties need to be available for sequential and parallel workloads. Understanding system bottlenecks and exposing the hardware-agnostic properties of applications allow system architects to reason on what architectural decisions would best support the software. We address this problem by proposing a hardware-independent software analysis tool for sequential, OpenMP and MPI applications. The application signatures extracted with the proposed tool are used to analytically model the system performance and power consumption. We believe that our proposed solution will support the system-level design research to gain insights into the effects of fundamental design decisions and differentiate good from bad design points.

1.2 State-of-the-Art in Application Analysis and System Modeling

Workload characterization plays a crucial role in system design. A good understanding of workload properties can support decisions to match hardware to applications. Various techniques have been used to measure the inherent properties of applications and the interaction between algorithms and architectures, ranging from profiling of applications on specific hardware platforms using performance counters, to micro-architectural and full-system simulations and purely analytic methods.

Many studies such as [61], [17], [38] and [65] have characterized sequential applications in terms of CPU and memory usage, instructions per processor cycle, data and instruction cache misses, cache accesses, amount of bytes transfered between cache levels per instruction and memory bandwidth utilization. These detailed characterizations are accurate and valuable to understand the suitability of an existing hardware design to a specific set of applications. Moreover, they are useful studies to understand whether application performance could benefit from a certain architectural trend or not. However, such results are based on measurements performed on existing systems, thus the insights are usually limited to the experimental hardware.

Parallel applications are often characterized by tasks or communication graphs [44]. Each node represents a sequence of computations and the arcs represent data dependencies. Such graphs can be generated using complex software tools, e.g., Vampir [83], Tau [118], the HPC Toolkit [52], Extrae [3] combined with interfaces for use of the hardware performance counters like PAPI [100]. The communication calls of the parallel application are intercepted and information about the communication is extracted. Tools like PAPI are used to monitor the hardware counters to extract information about the performance of the compute segments between the communication calls. The workload measurement data is thus collected for offline analysis. While the communication properties of the workload are usually hardware-independent (the communication matrix, the amount of data exchanged between the communicating pairs, the size of the messages), the compute performance is hardware-dependent. Therefore, estimating the performance of the same application but run on a non-existing or existing system with different architectures of compute nodes is not possible. Empirical collections of data are valuable studies that provide accurate results to guide algorithm-architecture co-design, but they are dependent on the experimental hardware.

Analytic approaches are less accurate alternatives to empirical collections of performance data. These techniques are useful in guiding system design at a low-cost and they are fast alternatives to other performance estimation methods such as cycle-accurate simulation. However, they usually rely on simplifying assumptions (e.g., simplified architectures of processors, balanced computation and communication [54]) that do not necessarily hold in real-life implementations. Moreover, they are mainly applicable to kernels whose computational structure and communication phases are easily/manually deducible (e.g., the dependency graph of the application is easy to derive [54]).

An example of a comprehensive analytic analysis is presented by Kerbyson et al. [81]. The study is a characterization of the SAGE spatial decomposition algorithm, characterization used as input to a machine performance model. A-priori knowledge about the application is required to manually characterize it in terms of high-level operations, an operation being defined by the processing of a subset of cells in the grid. The characterization is based on manual analysis of the key structures of the algorithm and on information pertaining to machine traces. The latter are used to estimate the actual execution time of the high-level operations. Thus, the compute time of an operation is not analytically estimated, but obtained via measurements of existing machines, similar to the empirical collections of performance data.

Other analytic approaches are based on Amdahl's Law [20], [69]. Over the decades, they have been successful techniques for evaluating the performance and scalability of parallel applications. However, performance is estimated assuming specific values of processor performance metrics (specific values of, e.g., cycles per instruction), thus again the estimated execution time is hardware-dependent. In addition, manual analysis of the algorithm is necessary to quantify the application compute and communication behavior, e.g., in terms of number of processing operations or number of data transfers through the network.

In between analytic models and empirical collection of data stand simulators. Examples of frequently used micro-architectural simulators include [39], [27] and [45]. They achieve emulation speeds for a single processor of tens or hundreds of millions of instructions per second. In general, regardless of the level of abstraction at which such simulators analyze the stream of instructions, a large parameter space exploration of processor designs is very time and resource consuming. For each processor configuration, a new run is required to estimate the performance. Similar challenges are faced by network simulators [98]. Each network topology is described by a set of parameters describing the interconnect structure of the switches and nodes. For each system size and set of network parameters, it is necessary to run a new simulation. For full-system simulators such as [25], [105] and [35] the feasibility of evaluating a large design space is even lower. If for example one would like to simulate an application with 64, 128, 256, 512 processes each with 1, 2 and 4 threads per process, each process-thread configuration on a parameter space of 5 topologies with 50 configurations per topology and of processors with 10 micro-architectural parameters and 5 values per parameter, 150,000 simulations would need to be run. Replacing the network and the processor simulation with analytic models would bring a significant speedup, especially for very large design spaces. This is important when designing a system at exascale.

We propose an alternative methodology to full-system simulators that is more time-effective especially for large hardware design spaces. The key of our approach is to decouple the application characterization from performance and power modeling. We characterize the application in an ISA (Instruction Set Architecture)-agnostic manner and load the application's properties into analytic models. These models parametrize the hardware resources and enable fast system design-space exploration. Even if the hardware-agnostic software characterization incurs an analysis overhead, as the analysis is hardware-agnostic, it needs to be performed only once per application, for each input size and number of threads or processes. Therefore, the time cost of the software analysis can be amortized across several subsequent performance evaluations of specific hardware systems.

We rely our methodology on using the hardware-agnostic application properties with analytic models to estimate the performance of a system. We focus on the early phases of system design and, by using analytic models, we provide researchers and engineers with means to explore a large design space in a short time span. In these early stages, the relative accuracy of different design points is important in order to gain insights into the effects of fundamental design decisions. This way system designers can differentiate between worse and better design points. For absolute accuracy, methods with lower abstraction layers, such as simulators, could be further used. However, as they run significantly slower than analytic models, only a selection of the large design space could be analyzed into more detail.

1.3 Research Questions and Contributions

This thesis aims to address multiple research questions and two case studies as follows.

For system design based on measurements (and tracing) on existing systems, we study how to analyze system bottlenecks by separately measuring the time that parallel applications spend not only in compute and communication, but also in inter-process data dependencies. This is relevant especially in the case of asynchronous MPI implementations, such as a representative benchmark of graph analytics, Graph 500 [4]. In such applications, waiting for a message typically takes the form of continuous polling until the message is received. With the existing tracing tools, the polling is either categorized as compute or communication time. However, in reality the time spent in polling is time not only due to delays incurred by the network, but also due to delays incurred on the processors that need to generate the message. We propose a high-precision profiling methodology of analyzing the time spent in compute, communication and inter-process dependencies in MPI asynchronous applications. We perform a thorough profiling analysis of one of today's most representative data-intensive benchmarks, Graph 500. We show that, when running it on a current top supercomputer, it can spend 80% of the time in communication, meaning that improvements in the messaging infrastructure of the system can significantly decrease the application completion time.

The remainder of the thesis is dedicated to studying a different modeling approach than that of predicting system performance based on bottleneck analysis or measurements of existing systems. Namely, we investigate if it is possible to perform early and fast design-space exploration of large-scale systems by (1) decoupling the software properties from performance and power modeling and extracting compute and communication signatures inherent to applications, and (2) loading the platform-independent software signatures into analytic processor and network performance models. While accuracy is important in system performance evaluation methodologies, we mainly aim to build models that preserve the ranking of systems based on their performance. To this end, the contributions are guided by the next research questions.

How to extract hardware and ISA-agnostic software signatures from sequential and parallel applications at native execution time? In prior art, the application characterization for processor modeling is typically based either on measurements of performance counters of existing architectures, or on measurements of micro-architectural-agnostic signatures of x86 binaries. Not only that they are hardware- and ISA-dependent, respectively, but both approaches apply only to sequential code implementations (similar to interpreter-based characterization frameworks). To analyze the performance of large-scale systems, it is necessary to characterize parallel workloads. We introduce a novel compiler-based instrumentation approach to analyze both sequential and parallel applications in an ISA- and hardware-agnostic manner at native run-time. We call the framework that implements our approach the Platform-Independent Software Analysis tool (PISA), a framework built on top of the LLVM compiler infrastructure.

Can platform-independent software signatures be used with analytic processor performance models to estimate execution time and how accurate is the combination of the two? We use PISA profiles with two different state-of-the-art analytic processor models: (1) a traditional model that estimates performance by assuming that the events occurring in a processor are independent, and (2) a more recent approach that takes into account the interactions between the processor events. We provide the first analysis of the accuracy of ISA- and hardware-agnostic software signatures loaded into such analytic processor models. We also study the accuracy of a power model that uses PISA-based software profiles.

We further investigate what other types of processor performance modeling a framework like

PISA can enable. We use PISA to study how to estimate branch misprediction. To this end, we model branch miss rate based on PISA branch traces using two methods that characterize the predictability of the branch behavior of an application: the state-of-the-art branch entropy and a novel max-outcome branch prediction method. (1) We show how accurately branch entropy models hardware branch predictor parameters and describe the first method to reverse engineer the global history size of a branch predictor. (2) We analyze the limitations of branch entropy and propose a method to derive analytic models of hardware branch prediction. (3) We propose an alternative method to branch entropy, that we call max-outcome, which is not only correlated with the measured branch miss rates, but it is also more accurate than branch entropy when compared with real measurements. Moreover, in contrast to the branch entropy method, the max-outcome approach models not only the history size of a branch predictor, but also the pattern table size.

Estimating the performance of the processor is not sufficient for predicting the performance of a large-scale distributed system. The network plays an important role in the overall system performance. Thus, the next question that we aim to answer is how to analytically model network performance in a fast and accurate manner. We introduce an analytic performance analysis method that takes into account a network specification and the communication pattern of a parallel application to analytically derive the injection bandwidth of a node sustained by a network. We apply the method to three classes of communication patterns, representative of HPC patterns (uniform, shift and 2-dimensional nearest-neighbor). In contrast to state-of-the-art simulations or mathematical max-flow formulations, our proposed bandwidth models provide fast means to perform early and fast design-space exploration across network configurations of any size at good accuracy rate of more than 90% in the majority of cases.

With analytic models of processors and networks, we further introduce the first methodology for large-scale systems performance prediction that uses hardware- and ISA-independent software profiles. To estimate the compute time, we load the software compute profiles into analytic processor models (unlike the state of the art typically based on hardware-specific application profiles and (pseudo) cycle-accurate compute simulators). To estimate the communication time, we employ linear models of the transmission time of a message, models that use our analytic bandwidth estimators in combination with network latencies. By combining the communication time with the compute time, we derive a method to estimate the execution time of a parallel application. Validation results with measurements on real supercomputers indicate that our methodology could be used for early design-space exploration studies. Indeed, for the Graph 500 benchmark we obtained a linear correlation factor across hardware designs of more than 0.92.

We conclude our contributions with two realistic case studies. We perform a design-space exploration of compute nodes for radio astronomy (SKA) algorithms and of compute nodes and network topologies for graph analytics (Graph 500). For the SKA case we also contribute with an application-specific ASIC/FPGA power model for SKA front-end applications. With

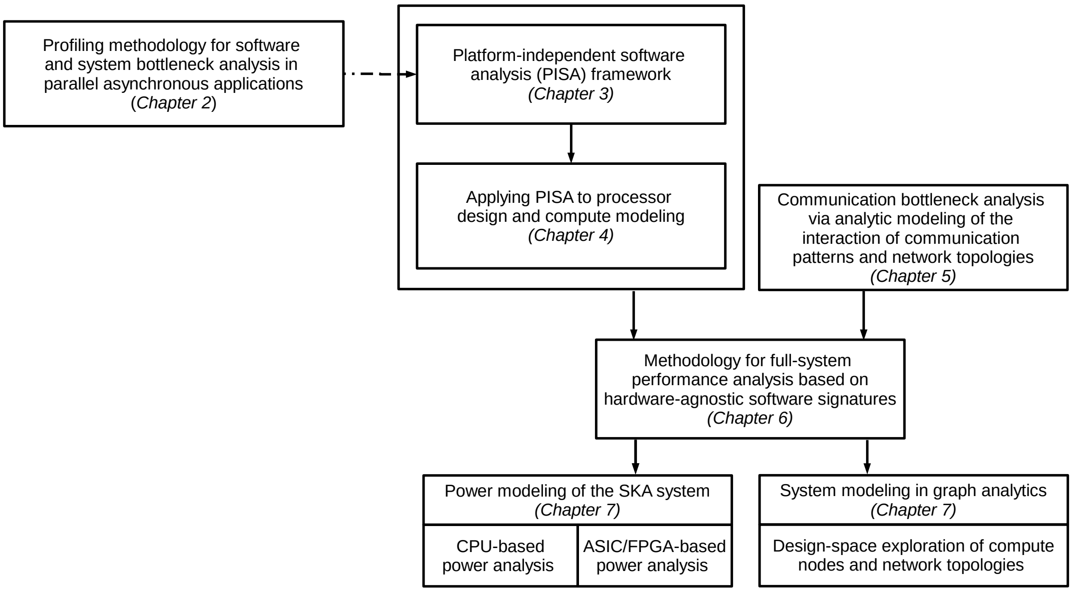


Figure 1.1 – A schematic representation of the contributions of this thesis.

our exploration studies we aim to (1) provide the SKA with preliminary guidelines on the design of its compute nodes, and (2) estimate the power consumption of components of the SKA pipeline assuming different hardware implementations. For the Graph 500 case study, we perform a power-performance trade-off analysis across different processors and network configurations for a system of 262,144 nodes.

1.4 Thesis Overview

The chapter organization of this thesis is summarized in Figure 1.1, whereas more detailed contributions of each chapter are presented in the next paragraphs.

Chapter 2

This chapter presents a high-precision profiling methodology for analyzing the time spent in inter-process data dependencies in MPI asynchronous applications. To understand system bottlenecks, the software profiling tools need to expose not only the time an application spends in compute and communication, but also in inter-process dependencies. We provide a methodology on how to analyze the time spent in data dependencies using state-of-the-art MPI profiling tools.

We apply this methodology to the graph analytics field, by conducting a wide array of tests on a high-performance computing system, the MareNostrum III supercomputer, for the Graph 500 benchmark. Our method shows that the most scalable MPI reference implementation of Graph 500 spends 80% of the time enabling communication (more than already reported estimations of 50% to 60%). We also provide a first detailed characterization of the communication pattern of the benchmark, analyzing it both in terms of data volumes between communicating pairs and behavior in time. The profiling method introduced in this chapter is useful for performance evaluation engineers and researchers that rely their software characterization and system bottleneck analysis on measurements of existing systems.

Chapter 3

While the previous chapter introduced an example of a thorough characterization analysis by applying a novel method to profile MPI applications on real systems, this chapter introduces PISA, an LLVM-based software instrumentation approach to characterize sequential and parallel C/C++ and Fortran implementations in a hardware-agnostic manner, at application run-time. We build this workload characterization framework with the goal to enable design-space exploration of existing and non-existing systems. The framework characterizes applications per thread and process in terms of instruction mix (scalar and vector instructions), instruction-level parallelism, memory access patterns, branch behavior and communication requirements. We apply our characterization framework on the Graph 500 and SPEC CPU2006 benchmarks. For these benchmarks, we provide a comparison between their characteristics extracted with our framework and real measurements of the same characteristics performed on x86 and POWER processors.

Chapter 4

This chapter is dedicated to show-casing PISA applications to processor modeling.

One case is using the hardware-agnostic software characterization results with analytic processor models to evaluate performance and power. We estimate the latter metrics for single-threaded applications from the Graph 500 and SPEC CPU2006 benchmarks using (1) a simple mechanistic first-order super-scalar processor model and (2) a more complex state-of-the-art processor model. We compare the results with POWER and x86 measurements. We show that loaded into state-of-the-art processor analytic models, the PISA characterization enables performance evaluation of processor micro-architectures with an accuracy of 34% on average across the SPEC CPU2006 and Graph 500 benchmarks.

The second use case focuses on one architectural component of the processor, i.e., the branch predictor. We analyze the branch behavior of an application using PISA-based traces and analytically estimate the branch prediction rate using linear models built from branch-entropy-based estimates and real measurements. We further devise a new branch-entropy-based method to reverse engineer the history table size of a branch predictor. We extend the state-of-the-art with a branch prediction characterization metric that models not only the predictor history length, but also limitations of how much information the predictor can store. The analysis we perform on branch prediction are high-level theoretical studies useful for guiding the design of future branch predictors.

Chapter 5

This chapter complements the previous ones with analytic performance models at the network level. To this end, we introduce a set of models for estimating the injection bandwidth of a node sustainable by a network. We quantify the link communication bottleneck due to packet network contention by analytically modeling the interaction of communication patterns (application property) with network topologies (hardware property). We derive models for three communication patterns (uniform all-to-all, nearest-neighbor and shift) and four network topologies (fat-tree, 2D HyperX, torus and full-mesh). We validate the proposed models by comparing them against state-of-the-art network simulations. The results show that the average accuracy of the analytic models is in the majority of cases above 90%.

Chapter 6

In this chapter we introduce a methodology for analytically evaluating the performance and power of large-scale systems. As input we use hardware-agnostic software profiles such as those extracted with PISA. Furthermore, we combine the communication models in Chapter 5 with the compute models in Chapter 4 and derive an analytic model for estimating the performance of large-scale systems. We validate the proposed methodology against measurements performed on real supercomputers for two applications: Graph 500 (as an example of application with uniform all-to-all communication pattern) and the NAS LU benchmark (as an example of application with nearest-neighbor communication pattern). To allow performance-power trade-off analysis for full systems, we also introduce a power model for both the compute and the communication components of the system. We use this power model to perform a design-space exploration of network topologies and compute nodes in Chapter 7.

Chapter 7

In this chapter we apply our performance modeling methodology to two fields: radio astronomy (Square Kilometer Array) and graph analytics (Graph 500). For the radio astronomy case, we perform a design-space exploration study for hardware compute nodes. We first provide an application-specific ASIC/FPGA model for estimating the power consumption of the SKA front-end processing system. We thus quantify how much more power-efficient an ASIC-based solution is than an FPGA implementation for three particular SKA algorithms. A variation of the FPGA power model has been transfered to the leaders of one of the SKA consortia. For the central processing stage of the SKA, we run design-space exploration studies of general-purpose CPUs using our proposed analytic performance evaluation methodology. For the graph analytics case study, we perform a full-system design-space exploration of compute nodes architectures and network topologies. We identify which combination of processing nodes and network interconnects best suit the most scalable MPI implementation of the Graph 500 benchmark for a network scale of 262,144 compute nodes.

Chapter 8 concludes the dissertation and discusses future work.

2 Profiling Methodology for Inter-Process Data Dependencies Analysis

Before introducing the components of a hardware-agnostic methodology for full-system performance estimation, we investigate in this chapter hardware-dependent time profiling tools for parallel (MPI) software. These tools are often used by system designers that build or optimize their systems based on profiling measurements of existing hardware. We apply a representative of such tools on the reference implementation of one of the most popular graph analytics benchmark, i.e., Graph 500 [4], run on a Top 100 supercomputing system [2].

Such profiling tools are key to efficiently design systems, because they provide information about the performance bottlenecks. Is the software run on the given hardware compute or communication bound? If communication-bound, how much time is spent in data transfer and how much in inter-process data dependencies? This is relevant information for a system designer, because only the time spent in data transfer can be optimized by optimizing the interconnect fabric. As out-of-the-box profiling tools do not differentiate between data transfer and data dependencies, we propose a scalable profiling methodology that quantifies not only the time spent in compute and communication, but also the time spent in inter-process dependencies.

The communication characterization results presented in this chapter are used for validation of the communication pattern extracted with the hardware-agnostic software analysis tool presented in Chapter 3. Also, the herein methodology could be used to quantify the time spent in data dependencies when using as input hardware-agnostic MPI traces of compute and communication events. Indeed, the traces could be fed to analytic performance models to estimate the actual execution time of the compute and communication segments. Once the original hardware-agnostic trace is timed, the methodology presented in this section could be used to estimate a detailed time breakdown of an MPI application. This represents a relevant future work item of this thesis.

This chapter is based on the following Springer article: Anghel A., Rodriguez G., Prisacari B., Minkenberg C., Dittmann G. (2015) Quantifying Communication in Graph Analytics. In: Kunkel J., Ludwig T. (eds) High Performance Computing. ISC High Performance 2015. Lecture Notes in Computer Science, vol 9137. Springer, Cham. DOI: 10.1007/978-3-319-20119-1_33.

The contributions of this chapter are two-fold. First, we provide a detailed characterization of the communication characteristics of the Graph 500 benchmark run on a real supercomputing system. The characterization includes the first analysis of the communication pattern of this application. Secondly, we describe a profiling methodology for accurately quantifying the compute and communication bottlenecks, by measuring also the time spent in inter-process data dependencies.

2.1 Introduction

Today's systems generate large volumes of data that need to be transferred, stored, but most of all processed to gain insights. As an example, over the last years, social networks have experienced an exponential growth up to as much as one billion active users with an average of more than one hundred connections each [31]. The complex processing involved in the exploitation of these large data sets requires the use of distributed workloads executed on massively parallel systems. To guarantee a high level of performance for these workloads, it is essential to identify what their bottlenecks are, particularly whether their execution is computation or communication-dominated. In the latter case, optimization is especially of interest, as data motion is expected to become the dominant power consumption component in future HPC systems [55]. One solution to address this challenge is the development of mechanisms that better orchestrate the data motion through the system [41]. However, to implement such mechanisms it is necessary to first understand the application communication patterns.

In the context of data analytics workloads, one particularly relevant class of applications is graph-based analytics. Indeed, the large sets of data generated by social networks and business analytics are often modeled as graphs that need to be processed on large-scale distributed systems using scalable parallel algorithms. A representative of this class is the Graph 500 benchmark suite [102], which has been introduced to guide the design of systems envisioned to support data-intensive applications. The benchmark is designed to assess the performance of supercomputing systems by solving a well-defined graph problem, i.e., the breadth-first search (BFS) graph traversal.

We use the MareNostrum III supercomputer [2] to characterize the communication of the most scalable reference MPI implementation of Graph 500. We analyze the data exchange across processes and the variability of the communication-to-computation ratio with the problem size (scale, edge factor) and number of processes. To the best of our knowledge, this is the first study that shows the actual communication pattern of the benchmark, offering preliminary guidance for future application or network design optimization efforts. To improve the precision of our results, we introduce a profiling methodology enabling us to minimize the tracing overhead and adjust communication time for data dependencies. This alternative profiling methodology is useful for accurately quantifying the compute and communication bottlenecks.

The remainder of this chapter is organized as follows. We start with providing background information about the Graph 500 benchmark and its MPI-simple implementation (Section 2.2). In Section 2.6 we present a selection of related work. We continue in Section 2.3 with a brief description of the computing system and profiling tools we used in the benchmarking process, as well as with a description of the parameters we used for the Graph 500 experiments. The same section presents characterization results obtained employing out-of-the-box standard profiling tools. Section 2.4 introduces the custom application characterization methodology for communication profiling and presents the experimental results. We proceed in Section 2.5 with describing the experimental results obtained for the communication patterns study. Finally, we discuss the main take-aways of this chapter in Section 2.7.

2.2 Background on Graph 500

Graph 500 is a large-scale benchmark for data-intensive applications. The problem size is given by the *scale* and the *edge factor* parameters of the input graph. If scale is equal to V then the number of vertices equals 2^V and if the edge factor is equal to E then the graph has $2^V \cdot E$ edges. The benchmark implements two main kernels: graph generation and BFS. First, the benchmark generates a list of edges and constructs an undirected graph. The graph construction phase uses the Kronecker graph generator which is similar to the graph generation algorithm presented in [46]. 64 graph nodes are randomly selected and, for each node, the BFS tree with that node as root is computed. The BFS step is validated to ensure that the generated trees are correct. The output of the benchmark is the time necessary to perform the BFS and the number of traversed edges per second (TEPS). In this chapter, we focus on the analysis of the BFS kernel as in a graph analytics setting the graph itself would already exist.

Graph 500 provides 4 implementations of the BFS algorithm: simple, replicated-csr, replicated-csc and one-sided. All four implementations use a level-synchronized BFS algorithm, that is, all the vertices at a given level in the BFS tree are all processed before any vertex in a lower level of the tree is processed [15]. For the remainder of the chapter we will focus on the MPI-simple implementation, which, despite its name, is actually the most scalable among all the reference MPI implementations [124].

The MPI-simple version of Graph 500 implements the BFS algorithm as follows. Each MPI process uses two queues: a current queue (*CurrentQueue*) and a next queue (*NextQueue*). *CurrentQueue* hosts all the vertices that have to be visited at the current level. *NextQueue* hosts all the vertices that will need to be visited at the next level. At the end of a level, the two queues are swapped. In addition, each MPI process uses two arrays: *Predecessors* (list of parents) and *Visited* (to track if a vertex has already been visited or not). If an MPI process A needs to visit a vertex that is assigned to another MPI process B, then process A will send a message to process B (via the MPI Isend function), requesting that process B visit that specific vertex. The information passed via this MPI message will include the vertex to be visited, as well as the predecessor in A that triggered the visit. For optimization, multiple such requests can be aggregated in messages of a certain size. This *coalescing* size is a tunable parameter. In this study we use the default value of 4 KB.

Concerning the choice of the Graph 500 benchmark as representative for the graph analytics space, we are aware that other frameworks and implementations exist. Some of the notable examples are Giraph [1] and Graphlab [92]. Giraph is an iterative graph processing framework built on top of Apache Hadoop. It is written mostly in Java and uses sockets for communication. GraphLab is a distributed framework that allows easy implementation of asynchronous, dynamic and graph-parallel algorithms, built on top of an RPC library. We have nonetheless limited our analysis to Graph 500, because our current tools are implemented on top of MPI and as such are incompatible with the other two frameworks. As future work, we are looking into adapting the same methodology for TCP socket communication as well as extending it beyond the data analytics space by applying it to other relevant HPC benchmarks, e.g., SPEC MPI2007 [10].

Last but not least, within the Graph 500 benchmark, several implementations are supplied, but we have chosen to focus on the MPI-simple implementation. This is because, while the other implementations are expected to behave better than the *simple* approach in a small-scale environment with few nodes, they do not, as shown for example by Suzumura et al. [127], scale to the system sizes that are now usual in practice in datacenter and HPC systems.

2.3 Out-of-the-Box MPI Software Characterization

Our objective is to understand the application performance bottlenecks by profiling its execution. Thus, we have instrumented the Graph 500 MPI-simple implementation and used the Extrae tool [3] to monitor the time the application spent in: (i) in MPI asynchronous point-to-point communication calls (calls to MPI Irecv, MPI Isend); (ii) in polling MPI Test calls; (iii) in MPI all-reduce calls; and iv) outside of MPI functions.

We executed the instrumented code on 4 to 64 nodes with a total of 16 to 256 concurrent processes, on the MareNostrum III supercomputer and obtained a set of preliminary results, presented in Subsection 2.3.4.

2.3.1 Benchmarking Platform

To benchmark Graph 500 we used a large-scale supercomputing cluster, MareNostrum III [2], currently ranked 57^{th} in the November 2014 Top 500 list [12]. The cluster consists of more than 3,056 compute nodes and 48,896 cores. Each compute node has two Intel®SandyBridge-EP E5-2670 20M processors, each with 8 cores running at 2.6 GHz, and is equipped with 32 GB of memory (DDR3-1600). Nodes are interconnected using an Infiniband FDR-10 network.

2.3.2 Graph 500 Configuration

We downloaded the Graph 500 reference code [4] and compiled it on the supercomputing cluster using the Intel compiler version 13.0.1 and the Intel MPI 4.1.1 default version. The only modification we have made to the reference code was to insert events that mark the beginning and the end of each BFS (after the collection of the time statistics). In terms of problem size, we ran the benchmark for scales as small as 16 and as large as 26, while the edge factors used range between 16 and 256 in successive powers of 2. We used a number of concurrent processes ranging from 32 to 256.

2.3.3 Tracing and Analysis Tools

To extract the MPI traces of the Graph 500 benchmark we use a light-weight tracing tool, *Extrae*, formerly known as *mpitrace*, developed at the Barcelona Supercomputing Center. The traces have been further processed using two additional tools: a GUI-based trace analyzer called *Paraver*, and *Paramedir*, a command-line analysis tool based on Paraver. The two tools can filter the trace data and visualize time-lines, compute statistics, generate traffic matrices, i.e., spatial traffic distributions etc.

The tracing library *Extrae* is implemented as an *interposition* library that intercepts the MPI library calls. The tool stores the minimum amount of information needed to, in a later (off-line) phase, match all the communication partners and generate a single trace file that comprises all the communication activity, including parameters of MPI calls. The library also provides an API allowing custom emission of user events. We employ this capability, in particular by using *Extrae*'s API to mark the entry and exit to the relevant code segments of the Graph 500 benchmark. As we are measuring application completion time, it is important to quantify the overhead of the tracing tool. In all our experiments, the impact of tracing on the application runtime did not exceed 15%.

2.3.4 Characterization Results

As an initial approach, we used Extrae to profile all MPI calls that the Graph 500 simple implementation makes. This does not require changing the code of the benchmark in any way. However, in the course of a complete run of the benchmark, there is an initial graph construction phase, after which multiple BFS computation steps, which are the ones we are interested in analyzing, alternate with validation steps necessary only for solution verification and estimation of the number of traversed edges per second. As such, for convenience, we did minimal changes to the code to signal to the tracing tool the beginning and end of actual BFS tree computations, by means of emitting custom events. Only two events were needed, each requiring a single tracing library call.

Using this straightforward approach, we were able to determine that during the execution of the benchmark, the application is performing one of four types of activities: (1) asynchronous

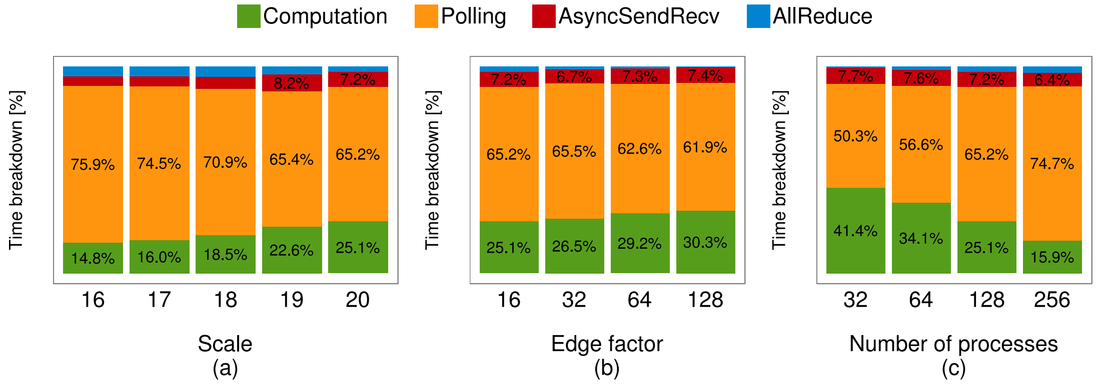


Figure 2.1 – Standard instrumentation. Single BFS execution time percentual breakdown for varying graph scales under a constant edge factor (16) and 128 concurrent processes (a), for varying edge factors under constant graph scale (20) and 128 concurrent processes (b), and for varying number of processes under constant graph scale (20) and edge factor (16) (c).
From [21] ©Springer International Publishing Switzerland 2015.

point-to-point communication (MPI Isend or MPI Irecv); (2) polling of pending asynchronous messages (MPI Test); (3) calls to MPI all-reduce; (4) computation, quantified as the time spent outside of any MPI calls.

The results obtained using standard characterization are outlined in Figure 2.1. The figure shows the percentual breakdown of the application execution into each of the activities enumerated above. The three sub-figures present the impact on this breakdown of three main application parameters: the scale and the edge factor (which define the problem size) and the number of concurrent processes executing the application. Several insights can be extracted from these results. First, the majority of the execution time of the application is spent in either computation or waiting for data to arrive from other processes (polling). In general, polling time dominates, accounting for more than 50% of the time in all scenarios and for more than 60% of the time in typical scenarios (where a reasonably high number of concurrent processes are employed). Second, there are a few clear trends of breakdown evolution with the three application parameters: (i) as the scale increases, computation becomes more important, (ii) as the edge factor increases, computation becomes more important, and (iii) increasing the number of concurrent processes significantly decreases the importance of computation. Overall, in a scenario where increasingly large problems would be solved by means of increasing the computational resources, the problem will remain highly communication-bound.

2.4 Custom MPI Software Characterization

By performing a more detailed data-dependency analysis, we present in this section a new set of characterization results. Using the Paraver and Paramedir analysis tools [86] on the resulting execution traces, we were able to quantify the proportion of the overall completion time that was spent waiting for communication to complete, performing computation or waiting due to data dependencies even for large problem sizes. We start this section with describing our methodology for extracting the time spent in inter-process data dependencies.

2.4.1 Methodology for Inter-Process Data Dependencies Analysis

In an application which uses synchronous communication, waiting for data from another process is typically performed via a single blocking call (either a blocking receive or a wait), which is logged in a communication trace very efficiently. In contrast, in an application using asynchronous communication, waiting for data takes the form of polling, that is, MPI test calls that query the communication infrastructure multiple times unsuccessfully before ultimately receiving a confirmation of data being available. Logging every failed test in the communication trace can lead to it becoming extremely large as well as inducing a high tracing overhead (and thus a warped view of the application). However, particularly in the case of a communication-bound application, it is precisely the failed tests that convey the time spent waiting for data.

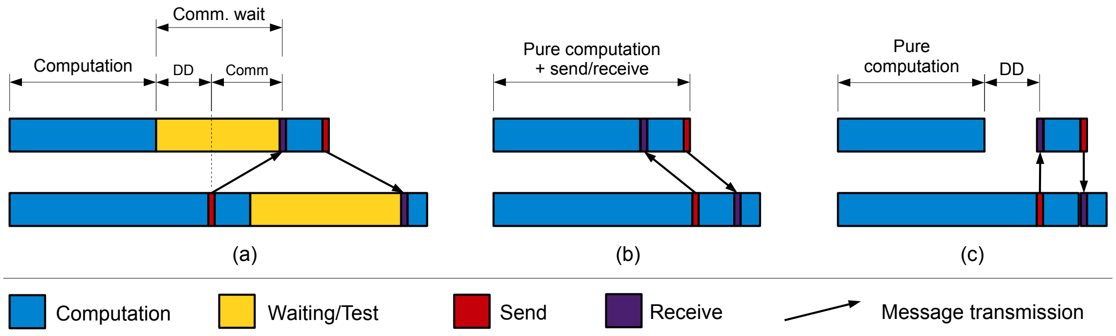


Figure 2.2 – Three types of traces for a minimal application with two concurrent processes. Fig.(a) shows a standard trace, Fig.(b) shows the trace where the communication is emulated from locally stored data, and Fig.(c) shows the trace were communication is emulated from locally stored data but the temporal data dependencies (DD) between MPI Isends and their corresponding MPI Irecvs are enforced.
From [21] ©Springer International Publishing Switzerland 2015.

To address this issue, we imagined the following thought experiment. Let us assume that failed tests are not logged in the trace, and instead time spent performing them appears as being outside of any communication event. This means that what now appears to be computation time in the trace is a mix of actual computation and polling. Should we be able to execute the application on the same system but ensuring ideal (zero-latency, infinite-throughput) communication, then whatever the completion time of this ideal run is, that would exclusively be the actual computation. To achieve this ideal setup, we replaced the standard MPI calls with custom calls that wrap the former and additionally have the capability to *record* traffic and *replay* it at ideal speed (i.e., incurring node-side delays such as memory copies, but not incurring any network-side delay). This allowed us to identify the time spent performing exclusively non-communication-related computation (Figure 2.2 (a) and (b)). The advantage of such an approach is not only that the trace itself is much more efficiently stored and collected, but more importantly, the actual computation is measured without any tracing overhead, and thus estimated much more precisely.

However, with this record-replay approach, removing polling time that is falsely registered as computation is not possible while at the same time maintaining the data dependencies between concurrent processes. Indeed, when executing a parallel workload, a process can be caught in a sequence of polling phases in two main circumstances:

1. the next steps the process has to execute depend on data in flight that takes a certain amount of time to arrive due to imperfections or simply inherent limitations of the underlying communication system, such as network latencies and contention; as this time component is communication-related, we call it communication delay;
2. the next steps the process has to execute depend on data that has not yet been sent by the corresponding source process; this time component stems from application-inherent data-dependencies between concurrent processes and we call it inter-process data dependencies, labeled as DD in Figure 2.2.

To be able to quantify the actual communication time, we need to partition the waiting time into the two categories above. For every message exchanged, the trace comprises both the source and the destination processes. As such, it provides all the necessary dependency information to achieve this partitioning. However, in order to be able to schedule the dependencies, we used another tool, called DIMEMAS [32], which is capable of ingesting traces captured with Extrae, replay them maintaining the semantic data dependencies and additionally model the inter-process communication for arbitrary levels of bandwidth, latency and network congestion. Using this tool, we were able to determine the application runtime in the absence of communication delays (Figure 2.2(c)). This runtime, coupled to the ideal runtime above and the real runtime, the latter two measured on the real system, are sufficient to allow us to identify with high accuracy the time the application spends in each of the following five types of activities: (1) asynchronous point-to-point communication; (2) calls to MPI all-reduce; (3) actual (as opposed to apparent) communication-related polling; (4) inherent data dependency related polling; (5) actual computation.

2.4.2 Characterization Results

Using this custom methodology, we re-ran the experiments presented in the previous section and obtained the results illustrated in Figure 2.3.

First, as expected, the results show that a significant amount of what appeared to be computation time was actually overhead due to tracing (particularly due to the inclusion of the

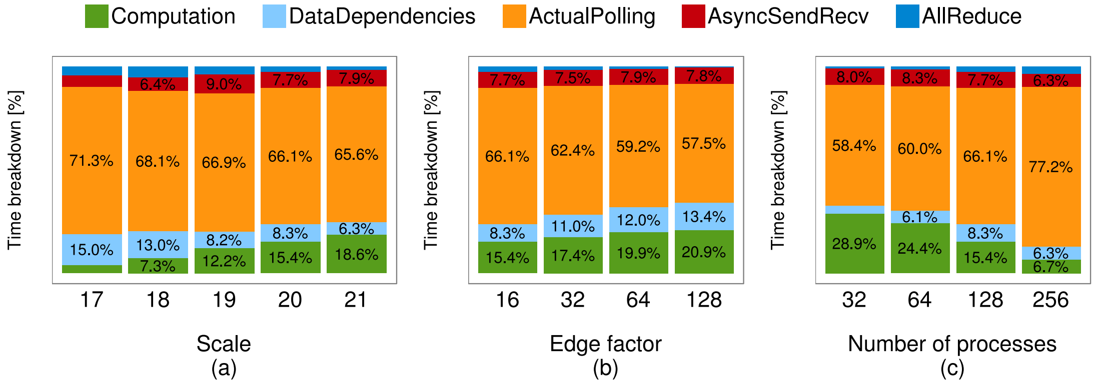


Figure 2.3 – Custom instrumentation. Single BFS execution time percentual breakdown for varying graph scales under a constant edge factor (16) and 128 concurrent processes (a), for varying edge factors under constant graph scale (20) and 128 concurrent processes (b), and for varying number of processes under constant graph scale (20) and edge factor (16) (c).
From [21] ©Springer International Publishing Switzerland 2015.

failed tests). While relative to the entire application execution, this overhead was moderate (< 15%), as it was accounted for practically entirely in the computation part of the trace, it made up a significant portion of that part. Indeed, while the previously identified trends are still present, actual computation is in fact approximately 40% smaller than what the standard instrumentation estimated.

Second, data dependencies make up a non-negligible part of the execution time, which now the custom characterization correctly identifies. In terms of trends, data dependencies seem to become relatively less important with the increase of the scale and more important with the increase of the edge factor. Also, as the degree of parallelism increases, their importance relative to the time spent computing increases significantly.

With the custom methodology and the resulting smaller traces, we are also able to handle larger problem sizes and degrees of parallelism. Figure 2.4 shows the results obtained for scales as large as 26 and 256 concurrent processes (the edge factor was set to the standard benchmark value of 16).

The figure shows that the previously identified trends hold, with communication remaining by far the dominating component, accounting for more than 80% of the total execution time (more than 75% of which is spent waiting for data).

2.5 Communication Patterns Characterization

In a parallel application implementation, optimizing the data exchange is key to improving the performance of graph-based analytics applications. Understanding the characteristics of

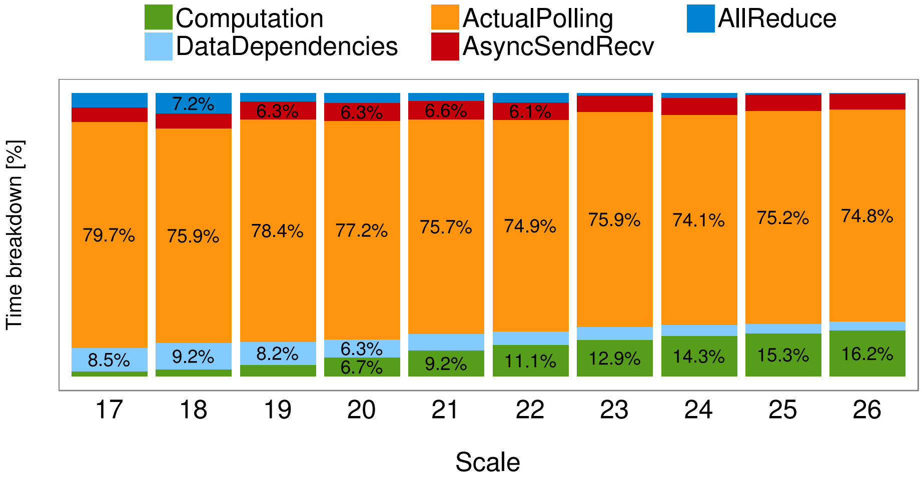


Figure 2.4 – Custom instrumentation. Single BFS execution time percentual breakdown for varying graph scales under a constant edge factor (16) and 256 concurrent processes.
From [21] ©Springer International Publishing Switzerland 2015.

the data exchanges is crucial to developing efficient systems and networks that enable these optimizations. Fortunately, the tracing and analysis tools we have at our disposal have a high enough granularity to allow data motion characterization. Indeed, from the communication traces, we were able to determine the amount of data that each (source,destination) task pair exchange, in time. The resulting traffic-matrix heatmap for a representative scenario (scale 20, edge factor 64 and 64 concurrent processes) for the entire duration of a single BFS is shown in Figure 2.5(a). Figure 2.5(b) shows the distribution of the amounts of data exchanged across all individual (source,destination) pairs. From both illustrations, one can see that the data exchange pattern strongly resembles uniform all-to-all communication. Indeed, the distribution has a standard deviation of only 8.6% around the mean and is almost entirely contained in an interval of 20% around the mean, with a very slight positive skew.

While these results suggest a uniform all-to-all traffic pattern, they are not sufficient to reach such a conclusion. A given (source,destination) pair could indeed exchange over the execution time of the program as many bytes as any other pair. However, the performance of the exchange and the communication pattern itself can vary strongly depending on how this global amount of data is aggregated into messages. Exchanging the 512 KB in numerous small messages or a few very large messages will lead to very different traffic signatures. To shed light on this aspect, we continue the characterization by extracting a similar heatmap (Figure 2.6(a)) and distribution across communicating pairs (Figure 2.6(b)) for the average message size. This analysis shows that the variability in the case of the message size is even smaller than in the case of the aggregated amount of data exchanged. Indeed, across communicating pairs, the standard deviation is only 0.7% around a mean of 3.75 KB per message. It should be noted that message size is a direct function of the (configurable) coalescing size parameter. For this study,

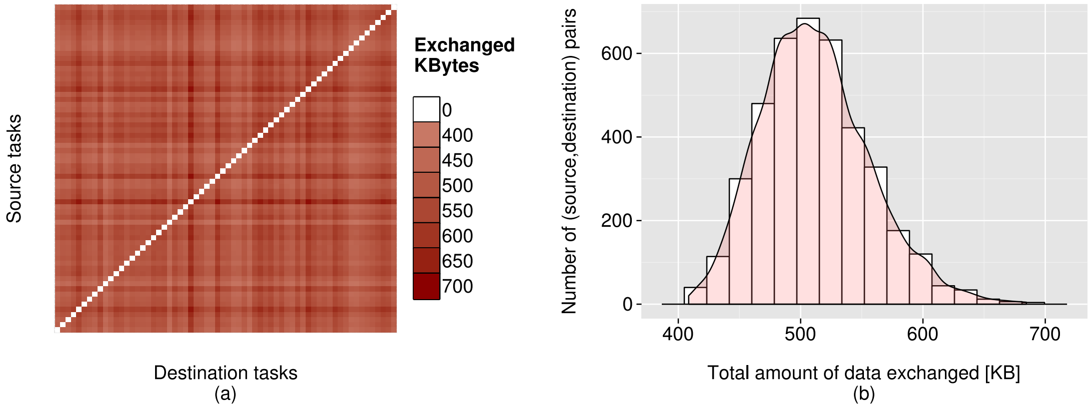


Figure 2.5 – Representative single BFS computation for scale 20, edge factor 64 and 64 concurrent processes. Figure (a) illustrates the traffic matrix between the processes and suggests that data exchanges are approximately uniformly distributed between all possible (source,destination) pairs. Figure (b) shows the actual distribution of (source,destination) pairs communication volumes across possible data amounts. The volumes are distributed approximately Gaussian around a mean of 512 KB with a standard deviation of only 8.6% and a slight positive skew.
From [21] ©Springer International Publishing Switzerland 2015.

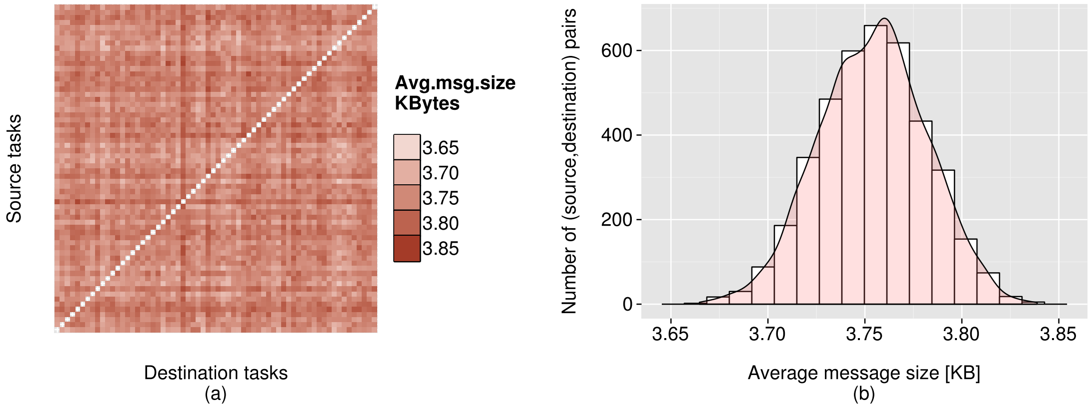


Figure 2.6 – Representative single BFS computation for scale 20, edge factor 64 and 64 concurrent processes. Figure (a) illustrates the distribution of the size of the exchanged messages across every possible pair of communicating tasks. The distribution is highly regular, with an average message size 3.75 KB and an extremely small variability (standard deviation is 0.7%), as illustrated in detail by the histogram in Figure (b).
From [21] ©Springer International Publishing Switzerland 2015.

we did not change the default 4 KB value. Choosing a different value for this parameter might have performance implications, but (for a reasonable range) it will not impact the conclusions of the data motion characterization.

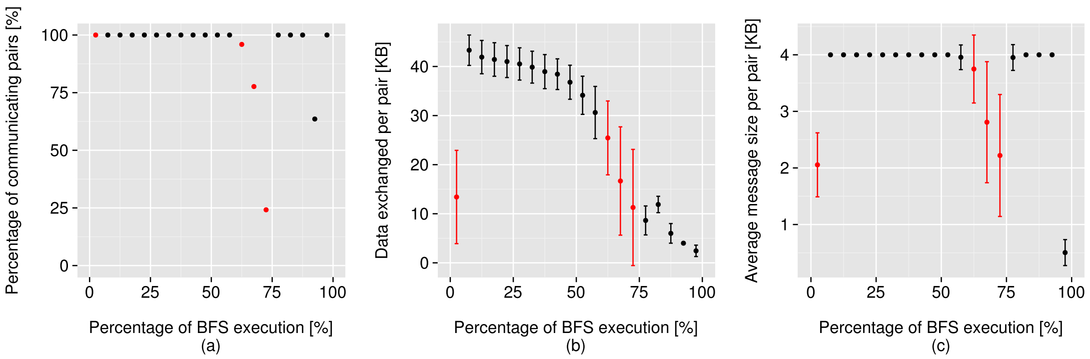


Figure 2.7 – Interval analysis of the communication pattern of a single BFS (scale 20, edge factor 64, 64 processes). The X axis represents the execution of the BFS in percentages. For every 5% of the execution time, we isolate the messages that are sent in that time window. For every time window: Figure (a) shows the fraction of all (source,destination) pairs that communicate in that interval; Figure (b) illustrates the mean and standard deviation of the amount of data exchanged in the interval across (source,destination) pairs; Figure (c) shows the mean and standard deviation of the average message size exchanged in the interval across (source,destination) pairs. For 80% of the intervals, the communication has the characteristics of a uniform all-to-all exchange.
From [21] ©Springer International Publishing Switzerland 2015.

Finally, even under low variability distributions of both aggregate communication volume and message size, a third aspect must also be taken into consideration when characterizing a potentially uniform all-to-all communication pattern, and that aspect is the distribution of the data exchange in time. To perform this analysis, we divide the entire runtime of the BFS into 20 intervals. For each interval, we perform the same two analyses that we performed before for the entire run, i.e., we compute the total and per-message communication volume and represent each resulting per-interval distribution by its mean and standard deviation. In addition, we also look at the number of active communicator pairs per time interval. The results are shown in Figure 2.7.

Figure 2.7(a) illustrates the percentage of active communicating pairs per time window. During 80% of the execution time, all pairs are in active communication. Figure 2.7(b) shows the data volume exchanged per communicating pair per time window. During 80% of the execution time, the amount of data per time window exchanged by an arbitrary communicating pair is similar to that of any other communicating pair (the standard deviation is lower than 20% in the majority of cases). Finally, the distribution of the message sizes is even more regular, as illustrated in Figure 2.7(c). During 65% of the time, there is no variability, i.e., every source is sending exclusively 4 KB messages, while during an additional 15% of the time the variability is very low (the standard deviation is lower than 20%). The remaining four windows (highlighted in red) manifest larger deviations from what would be expected from a uniform all-to-all exchange—only a subset of (source,destination) pairs communicate, and across that subset there are large variations in both the amount of data exchanged and the size of the messages used. However, we would argue that these intervals are not necessarily indicative of periods when a different communication pattern is taking place, but rather signs that imperfections/limitations of the communication infrastructure or load imbalance issues cause a limited subset of messages to experience long end-to-end latencies. Such tail effects can subsequently impact the application globally, leading to increased completion time and low network utilization.

In summary, these results suggest that system or network designs and optimizations targeting high-performance uniform all-to-all traffic have a high potential of positively impacting the communication performance of data analytics applications, and, consequently, the overall high performance of these communication-bound workloads.

2.6 Related Work

The work presented in this chapter lies at the intersection of two research topics: A) Graph 500 related characterization and B) communication profiling tools.

2.6.1 Graph 500 Characterization

Previous Graph 500-related research efforts, such as [124], [115], and [49], have implemented optimized versions of the benchmark, tested them on large-scale computing systems and reported their optimization techniques and performance results. In this chapter, we did not propose yet another optimized implementation of the benchmark, but we rather focused on an in-depth workload characterization of its most scalable MPI reference implementation.

In the characterization space, Jose et al. [79] report their findings on profiling the execution of the MPI-simple implementation version of Graph 500—tested with 128 processes and a problem scale of 26 and edge factor of 16—using unified MPI+PGAS communication runtime over IB. Excluding the synchronization cycles spent in the MPI *all-reduce* calls, a total amount of 60% of the total BFS time is predicted to represent communication. However, the runtime breakdown of one single problem size may not be sufficient to understand how the communication varies across different problem sizes and number of processes.

Suzumura et al. [124] perform a performance evaluation study of Graph 500 on a distributed-memory supercomputer for different types of implementations, including MPI-simple. The study reports profiling results for computation, communication and stall times. Even though the execution breakdowns are shown only for the replicated-based implementations (*replicated-csr* and *replicated-csc*), the authors expect the MPI-simple implementation to have similar performance characteristics as *replicated-csc*. The results show that communication and stall times account for less than 50% of the total execution time, even when increasing the number of nodes from 1 to 64.

In this chapter, we analyzed the breakdown of the execution time of the MPI-simple implementation, but across multiple scale, edge factor and number of processes values. This allowed us to identify trends in the execution breakdown of the benchmark. We showed that the communication time might be underestimated and that communication represents in some cases more than 70% of the total execution time. We also performed an in-depth analysis of the MPI-simple communication patterns across processes.

2.6.2 Profiling Tools for Parallel Applications

Crovella et al. [53] propose a methodology for measuring and modeling sources of overhead in parallel Fortran applications. The sources identified are load imbalance, insufficient parallelism, synchronization loss (defined as processor cycles spent acquiring a lock or waiting for a barrier), resource contention or communication loss. We propose a scalable method to achieve a similar breakdown by distinguishing within the communication loss between time spent at the destination waiting for data to be sent by the source – which we call inter-process data dependencies– and time spent waiting for data already in flight – actual communication delay. Furthermore, we use our proposed method to perform measurements for the Graph 500 benchmark.

A first step towards extracting characterization information from a parallel application is the use of profiling tools. The information provided by these tools is useful to break down the execution time of the application into time spent in communication and computation. For MPI applications, a typical way to do so is by interposing instrumentation functions between the MPI library functions and the MPI calls of the application. Such profiling tools will intercept application calls, particularly those involving communication: sends, receives, collective operations. This allows keeping a trace, generally in memory with regular flushes to disk, of the communication activity of the application. Some tools are able to provide more detail, such as performance counters before and after each profiled call, more information about the protocols or parameters with which the communication took place, or the user function from which the communication primitive was called etc. Moreover, these tools are able to dynamically profile with more or less overhead, re-compiling the application not being necessary. Some examples of profiling tools are: Vampir [83], Tau [118], the HPC Toolkit [52], or Extrae [3], among many others. For this work we will use Extrae, but we could have used any other tool, as we did not require any special feature or modification of the code of Extrae.

2.7 Conclusions

In this chapter, we performed an in-depth characterization and data motion analysis of a representative application in the graph-based analytics space. To this end, we chose to analyze the Graph 500 benchmark suite, which is widely considered a key application in assessing the performance of supercomputing systems on data-intensive applications. In particular, we focused on the most scalable implementation of the reference benchmark, the MPI-simple code.

Initial attempts of characterization using standard profiling exposed several limitations, mainly a high spatial and temporal overhead and a lack of support for data dependencies. Using a custom profiling approach, we addressed these issues and were able to target larger problem sizes and degrees of parallelism, up to scale 26 and 256 concurrent processes, while improving the accuracy of the characterization.

One of the main take-away messages of this work is that the Graph 500 benchmark which is representative of graph analytics workloads is communication dominated. Indeed, we have shown that the vast majority (more than 75%) of the execution time is spent in polling operations that represent waiting periods for messages in flight. This means that improvements in the messaging infrastructure (such as network bandwidth and latency or workload-specific routing or process to node allocation) will translate directly into a proportional decrease of what we identified as polling. In turn, the application itself will benefit greatly.

Furthermore, we quantified how the characterization changes with the problem size and the number of concurrent processes. Most importantly we identified that communication becomes less dominant with the increase of the *scale*, but significantly more dominant as more computational resources (concurrent processes) are assigned to the application. Moreover, we managed to separately quantify waiting times due to data dependencies and show that they can become important for the high levels of parallelism characteristic of HPC systems.

In performing this characterization, we also addressed several issues that a tracing tool will encounter when profiling applications using asynchronous communication. These include minimizing the tracing overhead, reducing the trace size by avoiding the storing of failed MPI test events and perhaps most importantly by pinpointing polling time due to data dependencies. Indeed, to make sure that we do not erroneously account in the communication time inherent synchronization waiting periods between applications (time spent in one process waiting for a message that another process has not yet sent), we carefully isolated this portion of execution time in the form of a data dependency time. While for the purposes of this study we made use of the Extrae tracing tool, the conclusions are not characteristic of Extrae alone, but rather of the entire class of tracing/profiling tools, e.g., Vampir, Tau or the HPC Toolkit.

Last but not least, analyzing the spatial and temporal distribution of the data exchange, we identified that the dominating communication pattern of the benchmark is uniform all-to-all, opening avenues for workload-specific data motion optimization of the Graph 500 benchmark.

3 PISA: A Hardware-Agnostic Software Characterization Framework

If in the previous chapter we focused on hardware-dependent time profiling tools often used to build or optimize systems based on measurements of existing hardware, in this chapter, we propose a hardware-agnostic application characterization tool to be used in methodologies that aim at exploring a large number of configurations of existing or future hardware systems. We describe in detail a hardware-independent software analysis tool called PISA (Platform-Independent Software Analysis) that will be used in the subsequent chapters to enable processor and full-system performance evaluation.

The contributions of this chapter are four-fold. (1) We propose PISA, a modular LLVM-based hardware-agnostic software analysis framework applicable to both sequential and parallel applications. (2) We showcase part of the capabilities of this framework by characterizing an application representative of graph-based analytics, the Graph 500 benchmark [4]. 3) We compare application characteristics extracted with PISA for the Graph 500 and SPEC CPU2006 [122] benchmarks with real measurements performed on x86 and POWER8 processors. 4) We propose a generalized memory reuse distance metric to characterizing both the spatial and temporal memory locality of a program.

3.1 Introduction

Workload characterization plays a crucial role in system design. A good understanding of workload properties can support decisions to match hardware to applications. Various techniques have been used to measure the inherent properties of applications and the interaction between algorithms and architectures, ranging from profiling of applications on specific hard-

---

This chapter is an extended version of a Springer journal article, Anghel, A., Vasilescu, L. M., Mariani, G. et al. Int J Parallel Prog (2016) 44: 924. DOI: 10.1007/s10766-016-0410-0, which is an extended version of the following ACM conference publication: Andreea Anghel, Laura Mihaela Vasilescu, Rik Jongerius, Gero Dittmann, and Giovanni Mariani. 2015. An instrumentation approach for hardware-agnostic software characterization. In Proceedings of the 12th ACM International Conference on Computing Frontiers (CF '15). ACM, New York, NY, USA, Article 3, 8 pages. DOI: http://dx.doi.org/10.1145/2742854.2742859.

ware platforms using hardware performance counters, to micro-architectural simulations and purely theoretical methods.

For example, Ferdman et al. [61] investigate the interaction between scale-out workloads and x86 processors. Scale-out workloads operate on large data sets that are split across a large number of machines and serve large numbers of independent requests that do not share any state. The study shows that the memory bandwidth is over-provisioned and that memory-level parallelism in scale-out workloads is low. These types of studies provide accurate and valuable results to guide algorithm-architecture co-design, but they are limited to the experimental hardware.

Simulators and emulators [45, 25, 105, 35] are a viable alternative to empirical profiling studies, because of their flexibility of easily changing the hardware parameters of the simulated system. For each hardware design the applications are run through the simulator and performance results are collected for different hardware designs. However, this approach is time-consuming, particularly when exploring a broad design space.

At the opposite end of the spectrum, theoretical approaches for algorithm analysis such as [117, 54] are fast alternatives for guiding system design. However, they are usually based on simplifying assumptions like abstract algorithm models, thus, cannot capture the application complexity. This usually leads to a decrease in precision.

To bridge the gap between theoretical approaches and simulators, we propose enhancing the former with a more detailed application model. To this end, we introduce PISA, a platform-independent software analysis tool capable of extracting a signature capturing the application's complexity. The signature consists of a set of hardware-independent application characteristics that can then be fed to analytic hardware models [77] to perform fast and truly workload-specific performance evaluation. We use the LLVM infrastructure [88] to instrument the application's code and analyze it at basic-block and instruction granularities at native execution time. Even though we run the application natively, our proposed tool extracts hardware-agnostic properties. Hardware constraints can also be easily modeled, but even in that case the characterization remains independent of the execution platform. We believe that such a framework is a key tool for system designers, enabling them to understand application properties and ultimately system performance bottlenecks.

To analyze the inherent properties of an application in a hardware-independent manner, it is necessary to "run" the application on an *ideal processor model.* Hennessy and Patterson define in [68] the properties of such a processor. To name a few of these properties: (1) unlimited number of registers available, avoiding all write-after-write and write-after-read hazards – the SSA form represents this property; (2) unlimited number of execution units, so that an unbounded number of instructions can be executed simultaneously; (3) perfect branch prediction; (4) infinite instruction window width, and all memory accesses taking one clock cycle; 5) perfect address alias analysis (a load can be moved before a store only provided that the accessed addresses are not identical). This definition can be extended to an *ideal* parallel machine as follows: (1) the machine is a collection of *ideal* processors; (2) the processors are connected via an *ideal* network with unlimited bandwidth and zero latency; 3) there is zero stack overhead for libraries handling message-passing routines.

As an illustrative case study, in this chapter, we apply our methodology to an application representative of graph-based analytics, the Graph 500 benchmark [4]. This benchmark was introduced to guide the design of systems for data-intensive applications. It is designed to assess the performance of supercomputing systems by solving the breadth-first search (BFS) graph-traversal. The large sets of data that social networks and business analytics usually generate are often modeled as graphs. The scale of the data that such applications have to handle precludes storing the entire graph on one single node. Therefore, large-scale distributed computing systems are required, together with scalable parallel algorithms, to efficiently process the graphs.

The remainder of this chapter is organized as follows. In Section 3.2 we present the main contribution of this chapter, the hardware-agnostic instrumentation methodology. We continue in Section 3.3 with a description of the characterization metrics that are extracted with the proposed characterization framework. The same section presents a novel approach to characterizing the spatio-temporal memory locality of an application. Section 3.4 illustrates the framework capabilities (1) to enable multiple ILP analyses and (2) to understand the application memory access patterns and branch behavior. We proceed in Section 3.5 with validating the application characteristics extracted with PISA with the same characteristics measured on real systems. Finally, we discuss the related work in Section 3.6 and conclude in Section 3.7.

3.2 Instrumentation Methodology

In this section we start with providing background information on the LLVM compiler infrastructure. Then we present in detail the instrumentation methodology.

3.2.1 The LLVM Compiler Infrastructure

The LLVM infrastructure [88] compiles application code to an intermediate representation (IR) form (or bitcode) used for optimization before linking and code execution. The LLVM IR is low-level, language- and target-independent and uses a RISC-like virtual instruction-set architecture (ISA) with an unlimited number of virtual registers in single static assignment (SSA) form. The LLVM IR provides explicit data flow in SSA form and programmatic access to the control-flow graph (CFG). The application code in IR form is called a *module* and contains functions and global variables. Each function is a set of basic blocks, and each basic block is a set of instructions ending in a control-flow instruction.

During the LLVM compilation process, a set of optimization passes can be applied to the IR of

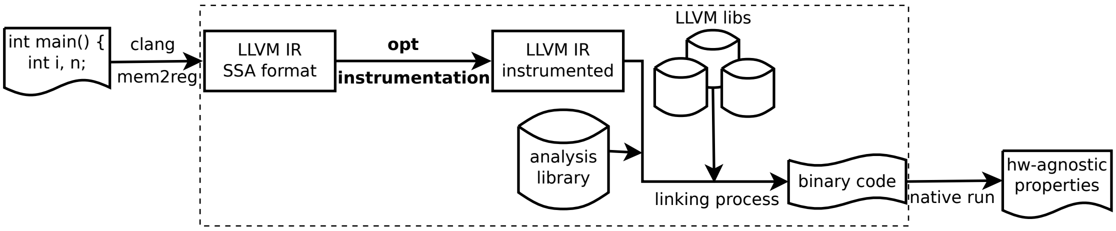


Figure 3.1 – Overview of PISA.
From [22], [23].

an application. The optimization passes are invoked by the *opt* command-line tool, which can load customized passes as plug-ins from dynamic library files. After optimization, the application is executed natively or using an interpreter.

3.2.2 Characterization Framework: Coupled Design

To ensure hardware independence, PISA analyzes the IR instruction flow rather than the assembly code running on a hardware platform. Even though LLVM provides direct access to the control flow graph of the program, the actual stream of executed instructions cannot be determined without interpreting or executing the program. PISA performs hardware-agnostic software analysis at native execution time. The output of PISA is a set of hardware-agnostic application properties that we call the application signature and serves as a software model.

Figure 3.1 shows an overview of PISA. The input is the source code of a program, e.g., in C/C++, which is compiled by the LLVM front-end, e.g., clang/clang++, to emit LLVM-IR bitcode (`.bc` files). This bitcode is transformed into SSA format by running the *mem2reg* optimization provided by LLVM. Next, PISA uses the LLVM optimization support, *opt*, to instrument the SSA bitcode with function calls to a modular external library that implements different workload analyses, such as instruction mix, memory access pattern, instruction-level parallelism, communication pattern etc. Finally, the instrumented bitcode is linked with the LLVM libraries and our external library to generate the final binary code. The binary code thus created is then run natively, and the program is analyzed. The native execution guarantees that our instrumentation collects information about the actual instruction flow of the input program.

An example of an instrumented LLVM IR code is shown in Figure 3.2. The bold lines represent code inserted in the instrumentation phase. This phase consists of inserting library calls in the LLVM IR in SSA format as follows:

- *init_library* - inserted only once at the beginning of the *main* function to pass the LLVM IR module bitcode of the program and various options (e.g., analysis types) to the external library;
- *analyze_bb* - used to notify the library about the next basic block that will be executed; it

```
        ; schematic LLVM IR
        @.str = “%d\n”
        @0 = “; ModuleID = ...”

        @main #0 {
        entry:
           %0 = getelementptr(@0, 0)
           call @init_library(%0, 0)
           call @func_address(0, @main)
           call @analyze_bb(0, 0)
1.           %x = alloca(10)
2.           %idx = getelementptr(%x, 0)
3.           store(42, %idx)
           call @update_vars(0, 0, 4, %idx)
4.           %idx1 = getelementptr(%x, 0)
5.           %tmp = load(%idx1)
           call @update_vars(0, 0, 6, %idx1)
6.           %y = getelementptr(@.str, 0)
7.           %call = call @printf(%y, %tmp)
8.           ret 0
```

Figure 3.2 – Instrumented LLVM IR in SSA form.
From [22], [23].

is inserted at the beginning of each basic block; each basic block is identified by a unique *(f,b)* pair, *f* being the function index and *b* being a basic block index inside function *f*;

- *update_vars* - used for passing the real memory address used by load and store instructions to the library; it is inserted after every such instruction;

- *func_address* - inserted at the beginning of the *main* function to notify the library about the real address of each internal function; our framework needs to differentiate between calls to internal functions and calls to external functions, i.e., functions whose code is not in IR form, such as functions from the `libc` library if its code is not compiled to bitcode; the real addresses are useful when a program uses pointers to functions.

- *exclude_function* - inserted at the beginning of the *main* function to notify the library to exclude certain functions from the analysis; this feature is used when the user does not want to analyze certain functions, such as reading input data from a file.

These library calls allow PISA to reconstruct the program's instruction flow as follows. The execution of each basic block is signaled to the external library. A basic block has a single entry point and a single exit point. This property allows the framework to process the sequence of instructions inside a basic block. Each instruction is processed in a *visit* library function that extracts different hardware-agnostic properties according to the type of analysis selected, and stores the information in different data structures. If the basic block contains load or store instructions, the library needs to ensure that *update_vars* is called after the instruction is processed. Furthermore, if the basic block contains a call to an internal function, the basic block execution will be interrupted and the execution will continue from the first basic block in the function called. In Figure 3.2 the *analyze_bb* call triggers the analysis of the instructions inside the first basic block. When a *store* instruction is encountered (instruction 3), the basic block processing is stopped and restarted by *update_vars*. The processing continues similarly,

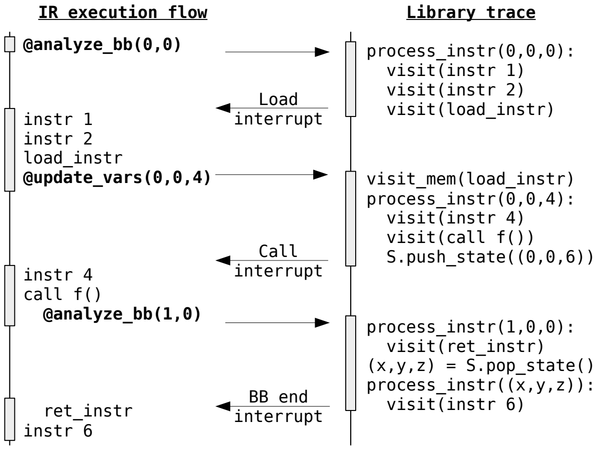


Figure 3.3 – PISA library events during application execution.
From [22], [23].

with restarts necessary for every load or store instruction or internal function call, until the end of the basic block has been reached.

For resuming the analysis after an interrupt an internal state is maintained. Figure 3.3 shows an overview of the library processing events that happen during the application execution. The state identifies an instruction as a triplet: *(function_id, basic-block_id, instruction_id)*. The library provides a function for processing instruction flows, *process_instr*. This function receives a state and processes instructions through the *visit* library call until the end of the basic block or until it is interrupted because it encounters a call to an internal function or a load/store instruction. *analyze_bb* calls *process_instr* with *instruction_id* equal to zero. If a load/store instruction is encountered, the processing is stopped and will be started by the *update_vars* call that sends the real memory address to the library (via the *visit_mem* call) and calls *process_instr* with the corresponding state triplet. If a call to an internal function is encountered, then the current state is pushed to a stack and the processing is stopped. The processing will be started again by an *analyze_bb* call from the new function. When the processing step reaches the end of the basic block, it checks whether the stack is empty. If the stack is empty, then the processing is stopped and will be continued by a new *analyze_bb* call. If the stack is not empty, the top state will be extracted from the stack and *process_instr* will be called with that state.

External libraries called by an application can also be analyzed using the same approach by compiling them to LLVM IR and linking them together with the application bitcode. In this case, both the application and external library code is instrumented and analyzed at run-time. Otherwise, without recompilation, PISA can be used to count how many and which external library calls have occurred at run-time. The user can use this information to compare the number of such calls with the number of instructions analyzed in the application and decide whether the number of instructions spent in external libraries is significant or not. In this work we have not analyzed the LLVM IR of the external libraries for two reasons. First, we wanted to analyze the application characteristics independently from a particular external library implementation. Second, as stated later in Sec. 3.5, for the applications analyzed in the current chapter, the number of instructions executed in external libraries is very small.

The framework can also analyze multi-threaded and multi-process applications and outputs separate profiles for each thread and each process. In multi-threaded applications, the address space is shared across threads. For each thread, the framework keeps independent data structures that will be updated by the *visit* call. The framework knows which structure to update based on the thread ID. The bitcode instrumentation is identical to that of single-threaded applications. If a basic block is executed by multiple threads, then *analyze_bb* will be called by each thread context, and only the data structure corresponding to the calling thread ID will be updated. Multi-process applications have a separate address space for each of their processes. For such applications, there is no need to differentiate between the processes, because the data are kept privately and the calls will only access the address space of that process.

Depending on the type of analysis, e.g., memory access patterns, instruction-level parallelism, the code instrumentation incurs an execution-time overhead of two to three orders of magnitude relative to the non-instrumented code. However, as the analysis is hardware-agnostic, it needs to be performed only once per application for each input size and number of threads or processes. Therefore, the time cost of the PISA analysis can be amortized across several subsequent performance evaluations of specific hardware systems. Each application input configuration entails a new pass through PISA, however, even comparing the time of a single non-amortized such characterization pass plus the time of an analytic performance model [77] versus the time of existing simulation approaches, our method can reach analysis speeds of several MIPS (millions of IR instructions per second) which is comparable or even one order of magnitude faster than the simulation method [45, 25, 105, 35].

3.2.3 Characterization Framework: Decoupled Design

In the PISA design described so far, the execution of an application thread and its profiling share a single thread. Therefore, each call to the analysis library stalls the application execution, incurring an execution time overhead. Moreover, in parallel applications, the overhead on one process may impact the other processes as well. For example, say an MPI process A performs an asynchronous MPI receive call and then enters a polling phase in which it continuously checks for the reception of a message from another MPI process B. If process B is stalled because of the analysis phase, process A might execute more instructions in the polling phase than necessary. Therefore, we propose a second PISA design that can reduce the analysis overhead. The IR instrumentation is still performed on the thread of the application. However, the analysis of the IR instructions is delegated to a separate process and, thus, the application

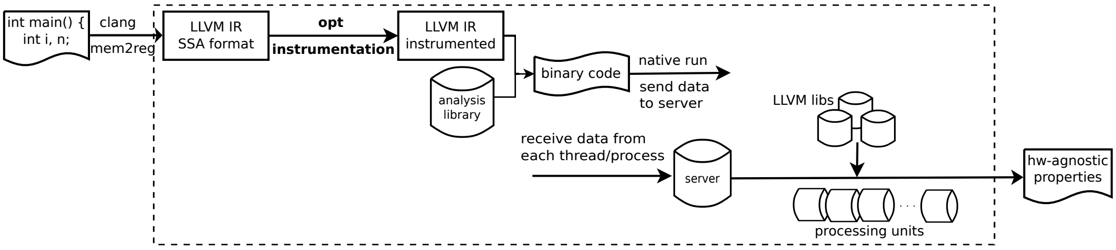


Figure 3.4 – Overview of PISA (decoupled version).
From [23] ©Springer Science+Business Media New York 2016.

can continue its execution without having to wait for the analysis to finalize. We call this approach decoupled and show an overview of this design in Figure 3.4.

The information of each instrumentation call that occurs on the application thread is sent to the analysis process in the form of a message. The analysis process uses the I/O event notification system call *epoll* to collect these application messages via sockets and stores their content in memory. For each OpenMP thread or MPI process A in the application, the analysis process spawns an independent thread that is responsible for performing the analysis for thread or process A. Not only is PISA capable of handling multi-process applications running over multiple nodes, but it also allows for the analysis component to be executed across multiple processes and nodes by providing multiple IP addresses to which the application MPI processes should communicate.

Both PISA versions can be used to perform hardware-independent workload characterization of sequential, MPI or OpenMP applications. Preliminary results show that the decoupled version can reduce the impact on the application execution time, leading to a slow down of only one order of magnitude over the native execution time. The analysis is still the bottleneck, but it has been offloaded to a separate process. The disadvantage of the decoupled architecture, however, is that the analysis process may require a significant amount of memory, particularly for long-running analyzed applications. This is due to the instrumentation messages potentially accumulating in RAM while they are waiting to be processed. Thus, both PISA designs are viable options for performing the characterization of an application and choosing one or the other depends on whether reducing the analysis overhead on the application completion time or the resources of the hardware on which the characterization is run is more critical.

3.3 Characterization Metrics

By analyzing the LLVM IR code in SSA form, PISA is able to extract a high-level application model that consists mainly of the instruction mix, the instruction-level parallelism, the memory access pattern and the branch behavior. For multi-threaded (OpenMP) and multi-process (MPI) applications, PISA extracts these metrics per thread (OpenMP) and per process (MPI).

The instruction-level parallelism is averaged across the entire execution of the application and consists of a set of numbers, each representing the instruction-level parallelism of a certain type of instruction (control, integer, memory, floating-point or all types). The memory access pattern has the form of a cumulative distribution and the predictability of the branch behavior is quantified by a number. For MPI implementations with point-to-point communication, PISA also extracts the amounts of data exchanged between the processes during the entire execution of the application.

PISA could extract the complete communication graph of the application, however, we restrict PISA to collect only a high-level signature of the application in order to enable fast design-space exploration using analytic hardware performance models. By removing the timing information from the application signature, we trade-off performance estimation accuracy for evaluation speed.

In this section we will focus on describing how PISA defines and extracts the different application properties, actual examples of PISA characterization results of real applications being presented in Section 3.4.

3.3.1 Instruction Mix

By analyzing the IR code of the application at run-time, PISA can extract the instruction mix, both scalar and vector instructions. Each instruction of an analyzed basic block is counted towards one of the LLVM instruction type categories: terminator, binary, bitwise binary, memory and miscellaneous. These are instruction categories that describe the IR format of the LLVM infrastructure. Mapping each type of instruction to real processor instruction set categories, e.g., integer, floating-point, control, load/store, is necessary to be able to use PISA's output as input to analytic processor performance models. Indeed, this mapping is required to quantify how much workload each type of functional unit in a real processor is supposed to execute. In the next paragraphs we will provide an overview of this mapping.

The LLVM terminator instructions end every basic block in a program and indicate which basic block needs to be executed next after the current basic block. These instructions typically produce control flow. Examples of such instructions are return and conditional/unconditional branches. We categorize these instructions as control, as they will be executed in a real processor by the control unit. These instructions, in addition to the function call instructions, are analyzed to characterize the control-flow behavior of an application at run-time.

The LLVM binary operators are used to perform most of the computation in a program. A binary operation takes two operands of the same type, execute an operation on them, e.g., addition or multiplication, and produce a single value. In the case of vector binary instructions, the operands are of vector data type. In that case the binary instruction produces a number of values equal to the number of elements in the vector operands. Binary instructions with integer operands are categorized as integer, whereas binary instructions with floating-point operands are considered floating-point instructions.

The LLVM logical binary operators are used to perform bit-twiddling in a program. They require two operands of the same type, execute an operation on them, e.g., shift or logical or, and produce a single value. The resulting value is the same type as its operands. All these instructions take integer or vector data types, thus, we categorize them as integer, as they will be executed in a real processor by the integer functional units.

The LLVM memory instructions read, write and allocate memory. The main two instructions to read/write from/to memory are load and store and we categorize them as load/store instructions, as they will be executed in a real processor by the load/store units. The LLVM IR load/store instructions are analyzed to extract the memory access pattern of an application at run-time.

Other LLVM instructions include function call instructions, conversion operations, comparison operations and address calculation operations. The call instructions are categorized as control, because a function call changes the control flow of the program. Conversion operations take a single operand and a type and perform bit conversions on the operand, e.g., truncate the operand to a different type or zero extend to a different operand type. The conversion operations that take as input an integer type are categorized as integer, while those that take as input a floating-point type are categorized as floating-point. Comparison operations return a boolean value. They take three operands. The first is a condition code to indicate what type of comparison to perform and the other two operands are the ones on which the comparison operation is applied. The comparison instruction that operates on integer operands is categorized as integer, whereas when the operands are floating-points it is categorized as floating-point. Address calculations are special LLVM instructions called *getElementPtr* which are used to get the address of a sub-element of an array data structure. It performs address calculation only and does not access memory. We categorize these instructions as integer.

3.3.2 Instruction-Level Parallelism

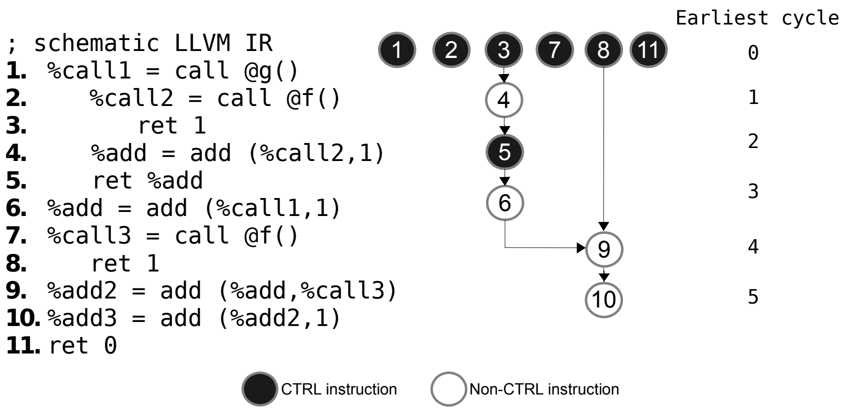
The instruction-level parallelism (ILP) can be measured only within a basic block or also across basic blocks. We focus on the latter as current processors enhance their performance by exploiting the application's ILP across basic blocks. To measure the ILP exhibited by a stream of instructions, our framework analyzes the dependencies between all the instructions and assigns to each instruction the earliest cycle number at which it can be executed on the machine model chosen. This is the minimum cycle number at which all dependencies of an instruction are satisfied on that specific model.

Multiple machine models have been proposed to analyze the ILP of an application. Lam et al. [87] propose a set of machine models that include several that are representative of real hardware. Similar to our approach, they examine the instructions from real programs and compute the available parallelism by enforcing true data dependency and control flow constraints associated with each machine model. The choice of a machine model, however, depends on the scope of the characterization. If the objective is to characterize the workload for a specific hardware architecture (e.g., a processor with 32 registers and two integer functional units), then that particular machine model must be assumed when calculating the ILP.

Our objective is to analyze a program's *inherent* compute and data motion properties in a *hardware-independent* manner. Therefore, we use the *ideal machine* described in Section 4.1. The ILP in such a machine is imposed by the actual data flows through either registers or memory. The resource (structural hazards) and storage (write-after-read and write-after-write) dependencies [28] are eliminated in our proposed framework since the LLVM virtual ISA uses an unlimited number of registers in SSA form. Thus, there are two types of dependencies left to analyze: true data and control dependencies. True data dependencies (read-after-write) happen when an instruction *i* produces a result that may be used by instruction *j*. Data dependencies through memory are difficult to detect in practice since two addresses may refer to the same location but look different a.k.a. aliasing. However, the LLVM IR overcomes this issue by allowing us to analyze true data dependencies only through registers. Even if the code uses pointer arithmetics, before the actual arithmetic operation, there will be a load to an LLVM register with the actual value of the memory operand used in the arithmetic operation. Thus, we can easily track all data and memory dependencies through the use of the LLVM registers. Control dependencies, on the other hand, happen when an instruction belongs to a basic block of a branch and, thus, it cannot be executed before the branch condition is evaluated, because until that evaluation, it is not known whether the instruction will indeed execute. If we assume perfect branch prediction, these dependencies are also eliminated.

Furthermore, in the *ideal* processor model, each instruction is assumed to be executed in one cycle. Finally, we also neglect the function call overhead, that is the overhead for setting up the stack frame, copying parameters and return address. Indeed, the overhead of such calling conventions would be too specific to the actual ISA, and our objective is to measure properties in a hardware-agnostic manner. The LLVM virtual ISA allows the framework to make this abstraction, because it does not consider any specific hardware-dependent calling conventions. The LLVM *mem2reg* tool promotes *alloca* instructions to register references, thus, it eliminates the stack overhead and any architecture-dependent calling conventions by using only registers.

Figure 3.5 shows an example of how the framework calculates the ILP of an instruction stream at run-time. The first step is to compute the dependency graph. Instructions that have no data dependency (1, 2, 3, 7, 8, 11) can be executed in cycle 0 because of the properties of the *ideal* machine model. Instructions that have a dependency only on instructions in cycle 0 (4) can be executed in cycle 1, and so on. If an instruction depends on instructions assigned to different cycles, it can be executed in the cycle following the maximum value of its predecessors' cycles. In Figure 3.5, every row of the graph corresponds to a different cycle. Data dependencies through memory are handled correctly through the *update_vars* mechanism described in Section 3.2.2. Note that dependencies on an immediate value returned by a function call are actually dependencies on the return instruction of that call and not on the call itself, e.g.,



```
; schematic LLVM IR
1.  %call1 = call @g()
2.     %call2 = call @f()
3.        ret 1
4.     %add = add (%call2,1)
5.     ret %add
6.  %add = add (%call1,1)
7.  %call3 = call @f()
8.     ret 1
9.  %add2 = add (%add,%call3)
10. %add3 = add (%add2,1)
11. ret 0
```

Figure 3.5 – Example of ILP calculation.
From [22], [23].

instruction 4 depends on instruction 3 and not 2. Otherwise, they are dependencies on the variable assignment.

We define the *span* of an application as being the maximum cycle number of any instruction calculated by our framework. Our target is to identify the minimum number of functional units in a processor core required to execute the application within the ideal span. To achieve this, we use a simple estimator, the *average ILP*, calculated as follows:

$$\mathrm{ILP}_{\mathrm{application}} = \frac{\mathrm{instructions}}{\mathrm{span}} \tag{3.1}$$

To determine the ILP per type of instruction (control, memory, integer or floating-point), we use the Eq. 3.1, where the number of instructions reflects the number of instructions of that particular type.

3.3.3 Memory Access Patterns

To understand the *inherent* memory access patterns of an application, PISA measures both its spatial and temporal locality patterns on the machine model described in Section 4.1, which assumes single-cycle memory access latency.

The temporal locality is measured by implementing the reuse distance analysis described by Zhong et al. [134]. The framework calculates the reuse distributions at application run-time and analyzes both the data and the instruction reuse distance distributions of a program. The reuse distance analysis consists of computing the number of distinct memory accesses since the last access to a given memory address, which is computationally expensive. To implement this operation, we use a splay tree because of its look-up or insert operations of `O(log n)`. Splay trees perform particularly well when the program analyzed exhibits high locality.

The reuse distance analysis does not supply information about spatial locality. Therefore,

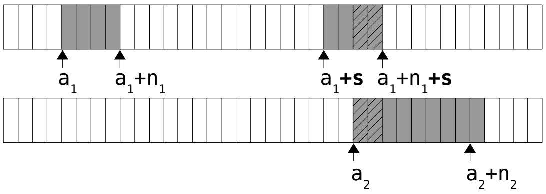


Figure 3.6 – Spatial locality definition.
From [22], [23].

PISA can additionally perform a more general memory pattern analysis that captures both the spatial and the temporal locality of a program, as follows.

PISA monitors all memory (load/store) instructions. In the resulting ordered list, it can assign consecutive indexes to instructions, starting with index 0. The index is a measure of temporality, as memory accesses with consecutive indexes will be executed consecutively in time. Every entry in the list is characterized by the following parameters: i—memory access index; a—memory access start address; and n—memory access length (in bytes).

Using this workload model, PISA calculates, given a current memory access, a measure of likelihood that another memory location is accessed a number of indexes in the future during the execution of the program. Figure 3.6 shows how we define spatial locality.

We want to define what it means for two accesses to occur at a distance of s in space. This is not as straightforward as subtracting the start addresses and checking whether the difference is s, because each instruction potentially accesses several memory locations at once. One way to define it is that at least one memory location accessed by the second instruction is at a distance of s in space from any location accessed by the first instruction. Formally, given a memory access (i, a_1, n_1), the framework computes $p(s,t)$, defined as the probability that the next t-th memory access $(i+t, a_2, n_2)$ will have the property

$$(a_1 + n_1 + s \geq a_2 \wedge a_1 + s \leq a_2 + n_2). \tag{3.2}$$

In this case, $p(s,t) \in [0,1]$ is defined for $t \geq 0$ and computed as follows. For each value of t, we consider every pair of accesses that are at a distance t from each other in the memory access list. Then, for every value of s, we count how many of these pairs satisfy the property given by Eq. 3.2. If that number is $k_{s,t}$, then $p(s,t) = k_{s,t}/(|L| - t)$, where L is the list of accesses and $|L|$ is its length. With $p(s,t)$ we define a joint measure of the spatio-temporal locality of a program, which can be represented as a locality heat-map.

By quantifying both the spatial and temporal aspects of locality, the approach we propose can, thus, supply useful input not only to cache memory design, but also to systems based on emerging technologies such as nonvolatile memory, systems that aim at bringing computation closer to data.

3.3.4 Branch Entropy

PISA can be used to measure the control flow behavior of an application. Indeed, PISA can monitor the outcomes of all conditional and unconditional branches in an application. Each branch is uniquely identified by three indexes, i.e., the function ID, the basic block ID and the instruction ID within the basic block. This identifier is the equivalent of the branch address in our branch behavior modeling approach.

Based on this branching information, PISA can estimate the branch misprediction rate of the application. To achieve this, PISA first computes the application's branch entropy as defined in Equation 3.3. Branch entropy, introduced by Yokota et al. [132], is an entropy-like measure of the predictability of the outcome of the next branch of an application, given the outcomes of the previous n branches, where n is called the branch history size. To obtain an ideal predictability estimate, Yokota et al. [132] take the limit of branch entropy when n goes to infinity. In contrast, we maintain the history size as a parameter of our estimator, the history size corresponding to the global buffer size of the hardware branch predictor.

$$\mathrm{BE}(n) = \mathrm{E}(n+1) - \mathrm{E}(n), \text{where} \quad \mathrm{E}(n) = -\sum_{x_n} p(x_n) \cdot log_2(p(x_n)) \tag{3.3}$$

To derive an actual misprediction rate from branch entropy, we use an approach suggested in the same paper [132] that basically consists in considering the next branch outcome as a stochastic binary process and applying the inverse binary entropy function to the branch entropy to obtain a misprediction rate. We explain in detail the concept of branch entropy in Section 4.4.

This is considered a lower bound on the misprediction rate under the constraint of having access to only a number of past branch outcomes equal to the chosen history size. The lower bound or ideal aspect may come from assessing the branch entropy of the application from its complete branching behavior.

3.3.5 Communication Patterns

To extract the communication pattern of an MPI application PISA needs to be able to quantify how much data each communicating pair of MPI processes exchanges during the execution of the application. PISA monitors the MPI library calls of not only MPI implementations with blocking (synchronous), but also with non-blocking (asynchronous) MPI communication calls. The instrumentation consists of an *mpi_update_db* PISA call inserted after each point-to-point, collective, initialization and finalize MPI call in the application's LLVM IR source code. The *mpi_update_db* call includes parameters with which the MPI call is executed at run-time. An example of LLVM IR code with an MPI instrumented call is shown in Figure 3.7.

The figure shows the instrumentation of an MPI_Irecv call. The parameters of the instrumentation call *mpi_update_db* are the following:

```
; schematic LLVM IR
…

@main #0 {
entry:
 …
 %call = call i32 @MPI_Irecv(i8* %tmp, i32 512, %struct.ompi_datatype_t* dt,
           i32 -1, i32 0, %struct.ompi_communicator_t* ct,
           %struct.ompi_request_t** %recvreq)

 call void (...)* @mpi_update_db(i32 90, i32 6, i32 0, i32 0, i32 2, i32 512,
      %struct.ompi_datatype_t* dt, i32 -1, i32 0, %struct.ompi_communicator_t*
      ct, %struct.ompi_request_t** %recvreq)
}
```

Figure 3.7 – Example of MPI PISA instrumentation.

- the index of the IR function from which the MPI call is called, e.g., 90;
- the index of the IR basic block from which the MPI call is called, e.g., 6;
- the index of the MPI call instruction within the basic block, e.g., 0;
- a PISA flag used to distinguish between MPI calls in Fortran and C, e.g., 0 (C code);
- a PISA MPI call code, e.g., 2 (MPI_Irecv);
- arguments of the real MPI call, e.g., for the MPI_Irecv in the Figure, the number of elements in the receiving buffer, the size of an element in the buffer, the processor ID of the other end-point of the communication, the communication tag, the MPI communicator, the real memory address of the MPI request.

To quantify the communication requirements of an MPI application and determine the process-to-process message exchange pattern, PISA uses parameters of the point-to-point MPI call: the number of elements in the send/receive buffer, the type of elements in the buffer and the source or destination process field. By multiplying the number of elements with the size of the element type, PISA quantifies the amount of exchanged data.

To determine the pair of communicating processes is straightforward in blocking (synchronous) implementations. The source or destination process index can be extracted from the arguments of the MPI call. However, in the case of non-blocking (asynchronous) communication, a special monitoring of the MPI_Test and MPI_Wait function calls is required. In a non-blocking (asynchronous) implementation, even though the process executes the MPI_Irecv call, the process does not wait in that MPI call to receive the data. Instead the MPI process executes MPI_Test calls that query the communication infrastructure multiple times unsuccessfully before ultimately receiving the actual message. The mapping between an MPI_Irecv call and its corresponding successful MPI_Test is possible via the unique MPI_Request of the MPI_Irecv call that is used by the MPI_Test to poll the reception of the message. The successful MPI_Test provides critical information for extracting the communication pattern of the application, the source (or the destination) of the message, information that is not necessarily present in the MPI_Irecv call arguments (see Figure 3.7, where the 4th argument of the MPI_Irecv call, which indicates the source of the message, is -1).

3.4 Characterization Results

In this section we illustrate PISA's capability to enable multiple types of ILP analysis that are useful for understanding system performance bottlenecks. Moreover, we show how PISA can be used to also understand application memory access patterns and branch behavior.

3.4.1 Experimental Setup

We ran our framework for all the characterization metrics presented in Section 3.3. We used LLVM and clang 3.4, OpenMPI v.1.8.1 and the Intel®OpenMP* runtime library (version 07.2014). As input, we used the Graph 500 benchmark suite [4]. Its problem size is defined by the *scale* (s) and *edge factor* (e) parameters of the input graph. For scale s, the number of vertices equals 2^s, and for edge factor e, the graph has $2^s \cdot e$ edges. The benchmark implements two kernels: graph generation using the Kronecker graph generator and BFS. 64 graph nodes are randomly selected, and for each node, the BFS tree with that node as root is computed. The BFS step is validated to ensure that the trees generated are correct.

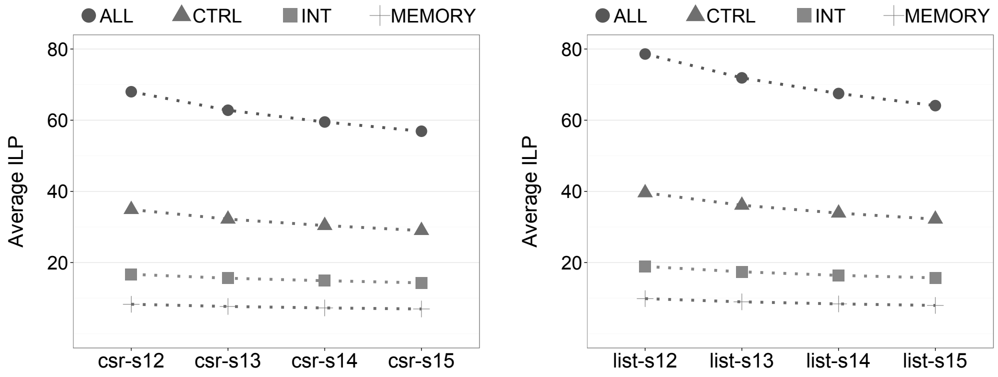
For the remainder of this section, we focus on analyzing the properties of the Graph 500 implementations shown in Table 3.1. *Seq-list* is a single-threaded list-based implementation, *seq-csr* is a single-threaded compressed-sparse-row implementation, while the last two are parallel implementations using either OpenMP (*omp-csr*) or MPI (*mpi-simple*).

Application	**Implementation type**	**Problem size**
SEQ-LIST	BFS - sequential	s12-15, e16
SEQ-CSR	BFS - sequential	s12-15, e16
OMP-CSR	BFS - OpenMP	s12-15, e16
MPI-SIMPLE	BFS - MPI	s12-15, e16

Table 3.1 – Graph 500 workloads used in PISA experiments.
From [23] ©Springer Science+Business Media New York 2016.

3.4.2 Instruction-Level Parallelism

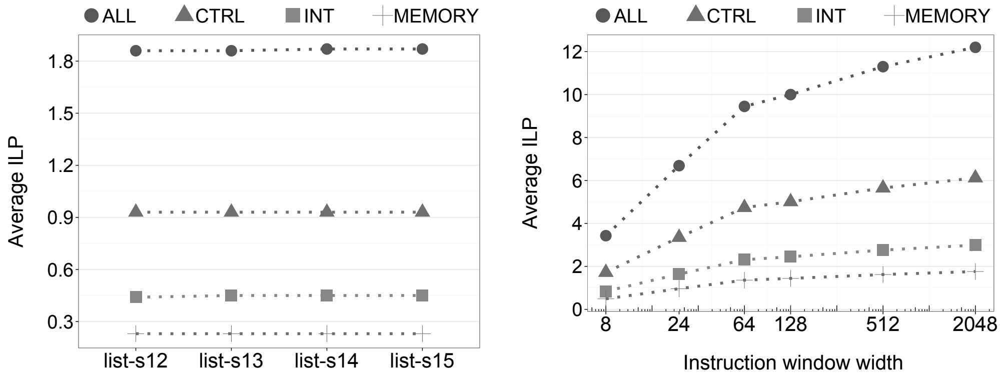
Figure 3.8 shows the average ILP per instruction type measured on the *ideal machine model* described in Section 4.1. The average ILP was measured for the *seq-list* and *seq-csr* implementations of Graph 500 for problem sizes ranging from scale 12 to 15. The *ALL* category represents the average ILP aggregated over all types of instructions, whereas the other three categories represent the average ILP for control, integer and memory (load, store) instructions, respectively. A category for floating-point instructions is omitted as such instructions occur seldom for this particular application. The results show that for the *ideal machine model* the Graph 500 implementations (and problem sizes) exhibit a high potential ILP of more than 55, with a higher ILP exhibited by control instructions (in the 30-40 range) and a moderate ILP for integer and memory instructions (in the 10-20 range). Furthermore, the two implementations exhibit similar ILP levels (with slightly higher values —as much as 15%— for the *seq-list* implementation), and there is a noticeable trend of ILP to decrease with the problem scale.



(a) Seq-csr implementation (scales 12-15). (b) Seq-list implementation (scales 12-15).

Figure 3.8 – ILP —ideal machine model.
From [22], [23].

Current hardware architectures are not able to exploit these high levels of ILP. This is due to hardware limitations that do not fit the *ideal machine model*. Among the most important limiting factors are the control flow constraints and the limited instruction window width. Our framework is able to take into account such hardware limitations and adjust its analysis.



(a) CTRL instrs. serialized (seq-list, scales 12-15). (b) Limited instruction window (seq-list, scale 15).

Figure 3.9 – ILP —ideal machine with hardware constraints.
From [22], [23].

Figure 3.9a shows how the average ILP changes when imposing a control-flow constraint on the *ideal machine model*, as defined in the *BASE* machine model in [87] which executes control instructions in order and at most one such instruction per cycle. We see that the average ILP has decreased considerably, falling below 2 (30 times lower) for the aggregated (ALL) ILP and below 0.5 for the integer and memory instructions. This is because the span increases considerably as a result of the serialization of the control instructions.

If we alternatively adjust the framework to take into account the limited instruction window size, we obtain the results shown in Figure 3.9b for *seq-list* scale 15. Similar results were obtained for the other scales and implementation. While not as extreme as in the case of control-flow constraints, a significant reduction in ILP of more than 6x for an instruction window width of 128 occurs also in this case.

While so far we have shown ILP results for sequential applications, our framework can also analyze parallel applications. We analyzed the ILP variability across the threads of the OpenMP implementation (omp-csr run with 8 threads or omp-8t) and across the processes of the MPI-simple implementation (mpi-simple run with 8 processes or mpi-8p), both analyzed for a graph scale of 15. The ILP was measured on the *ideal* machine with no hardware constraints. The results presented in Figure 3.10 show that there is a high variability across threads in ILP levels, with a standard deviation of 163 around a mean of 297 for the *omp-csr* implementation.

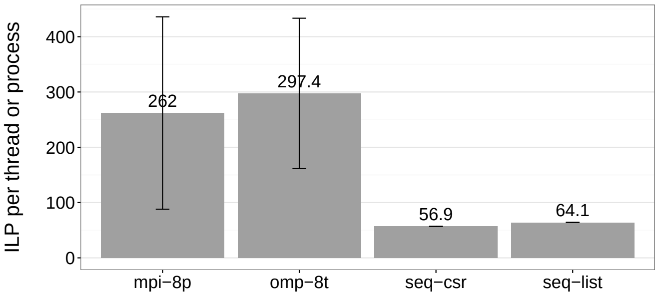


Figure 3.10 – Average ILP across Graph 500 algorithms (ideal machine - no hardware constraints). *From [22], [23].*

This variability can be explained by the stochastic nature of three sections of the application: (i) the graph generation, (ii) the partitioning across threads, and (iii) the root node selection for each of the 64 BFS phases, which causes each thread to perform different amounts of total work. Moreover, a large percentage of this variability is also due to the master thread, which performs a different set of tasks from all the other threads. If we restrict the analysis to just the worker threads, then the variability is smaller, with, e.g., a standard deviation of 100 around a mean of 254 for the same *omp-csr* implementation. Similar results have been obtained for the other scales and MPI implementation. It is also interesting to note that the parallel implementations exhibit higher ILP potential than the sequential ones.

We have also analyzed the ILP variability for the OpenMP implementation run with 2 (omp-2t), 4 (omp-4t) and 8 (omp-8t) threads and the MPI-simple implementation run with 2 (mpi-2p), 4 (mpi-4p) and 8 (mpi-8p) processes on the ideal machine with a limited instruction window

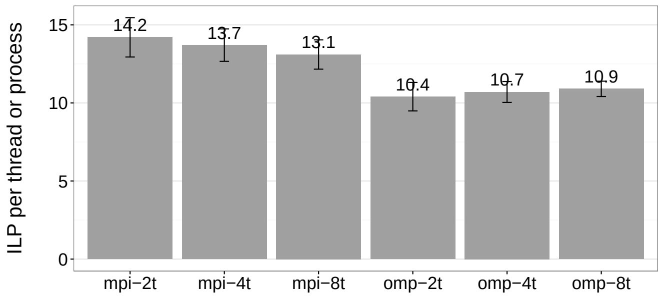


Figure 3.11 – Average ILP across Graph 500 algorithms (ideal machine - window size 54).
From [23] ©Springer Science+Business Media New York 2016.

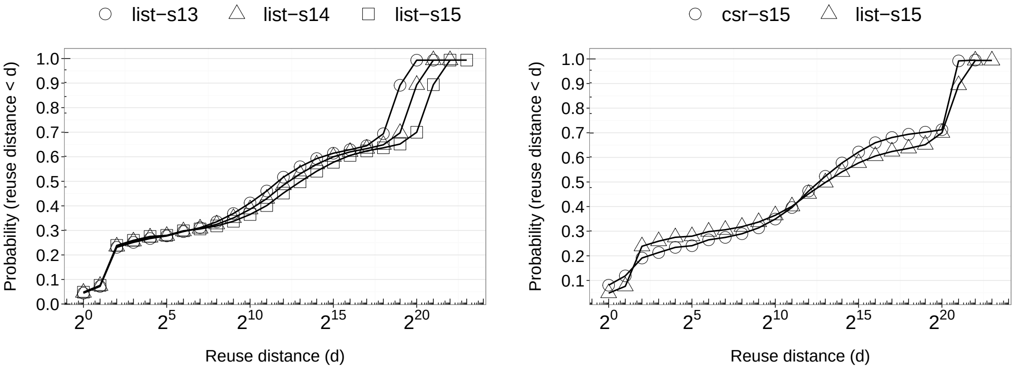
size (Fig. 3.11). For a realistic instruction window size of 54, the ILP variability across threads or processes and the ILP itself are at least one order of magnitude lower than in the case of an ideal machine with no hardware constraints. We also notice a decreasing ILP trend with the number of processes for the MPI implementation and an increasing ILP trend with the number of threads for the OpenMP implementation.

3.4.3 Memory Access Patterns

We start by showing results for the temporal memory access pattern of the Graph 500 implementations. Figure 3.12a shows the cumulative data reuse distance distributions for the *seq-list* implementation across multiple problem sizes. Figure 3.12b shows the distributions for *seq-list* and *seq-csr* for an exemplary problem size (scale 15).

The way we interpret these reuse distance distributions is the following. For a given reuse distance value *d*, the Y-axis represents the probability that between two consecutive accesses to the same memory address at most *d* distinct other memory addresses are accessed. This probability can be used to approximate the cache hit rate of an application run on a processor with a cache size of *d*, assuming least-recently-used cache eviction policy. The distributions in Figure 3.12a exhibit a similar reuse distance pattern. We note, however, that, as expected, the cache hit rate of the application decreases as the problem size (the size of the graph) increases, for all cache sizes. Figure 3.12b shows that *seq-csr* is slightly more cache-friendly than *seq-list* for typical cache sizes (*d* larger than 2^{12} = 4KB). For a cache size of, e.g., 262K (2^{18}), *seq-csr* exhibits a cache hit rate of 70%, whereas *seq-list* exhibits a cache hit rate of 63%. This is also expected as the *seq-csr* implementation uses data structures that are more efficient in accessing the memory than *seq-list*.

For multi-threaded applications, each application thread has its own data reuse distribution. To quantify the similarity of the temporal memory access patterns of the threads, we analyzed the variability of the distributions across the threads of an OpenMP implementation, by calculating the distance correlation between the distributions of each pair of threads as shown in Equation 3.4. The distance correlation of two distributions is calculated by dividing their



(a) Seq-list implementation (scales 13-15).

(b) Seq-csr and seq-list implementations (scale 15).

Figure 3.12 – Data reuse distance distributions.
From [23] ©Springer Science+Business Media New York 2016.

distance covariance by the product of their distance standard deviations and takes values ∈ [0,1]. The higher the distance correlation, the more similar the two distributions are.

$$\mathrm{dCor}(X,Y) = \frac{\mathrm{dCov}(X,Y)}{\sqrt{\mathrm{dVar}(X)\cdot\mathrm{dVar}(Y)}} \tag{3.4}$$

By applying the correlation metric, we found that there are two classes of threads, master and workers. The worker (non-master) threads exhibit very similar reuse distance distribution patterns. Indeed, the average correlation factor between pairs of such threads is 0.99, with a standard deviation of less than 0.9%. The master thread, in contrast, exhibits a fairly different distribution as the correlation factor between that distribution and that of any of the worker threads is only 0.33, with a standard deviation lower than 1%.

The temporal reuse distance distributions presented so far provide information only about the temporal memory access pattern of an application. They can be used to estimate the cache hit rate of an application run on caches with given size and line size.

To quantify also the spatial locality of an application, we generated locality heat-maps as described in Section 3.3.3. Before showing the locality heat-maps for the Graph 500 implementations, we first show-case the value of our heat-maps, by implementing a proof-of-concept algorithm that clearly exhibits spatial locality. We call this algorithm *customized random access*. We designed the algorithm as shown in Listing 3.1, where v is a vector of $N = 100$ elements. Every $t+1 = 15+1 = 16$ memory accesses the same vector element address is referenced.

```
m = 3000; N = 100; d = 10; t = 15;

for(int j = 0; j < m; j++){
    int increment = 0, sum = 0;
```

```
    for(int k = 0; k < t; k++){
        int random = ((double) rand())/(RAND_MAX+1E-6)
        var(k) = random*(N-d);
        sum = sum + v[var(k)];
    }

for(int k = 0; k < t; k++)
    sum = sum + v[var(k)+d+increment++];}
```

Listing 3.1 – Algorithm with spatial memory locality (customized random access).

Figure 3.13 shows how the locality of this program is represented using two commonly used approaches and our locality heat-maps.

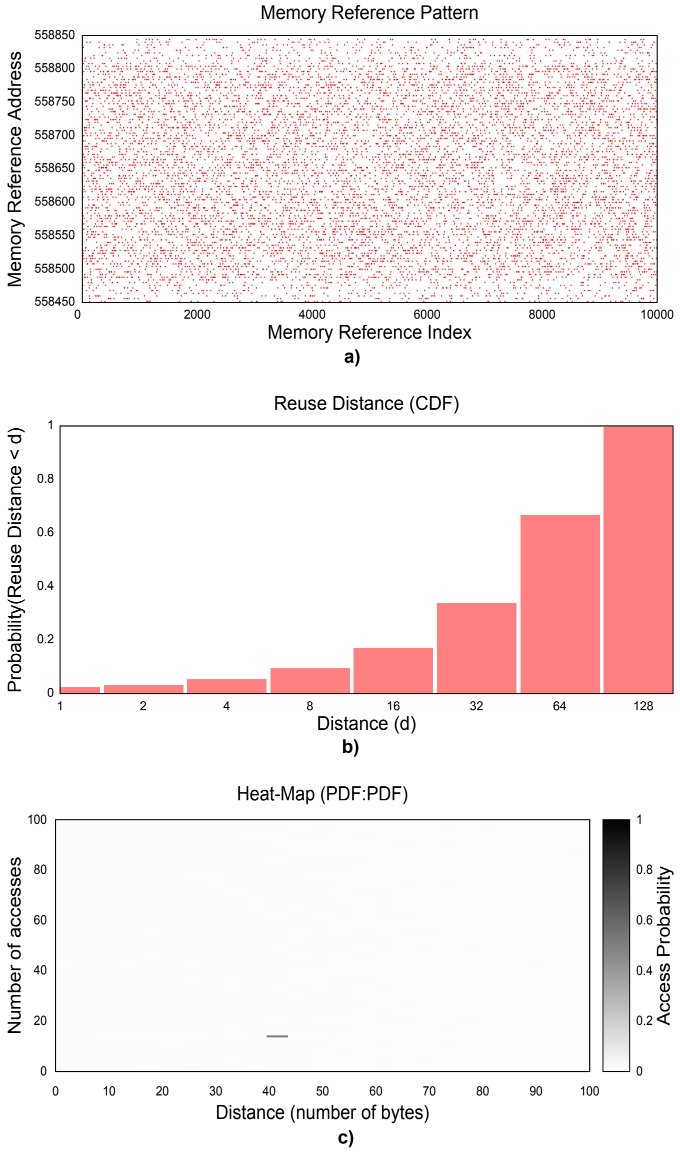


Figure 3.13 – Three locality representations for the *customized random access* algorithm.

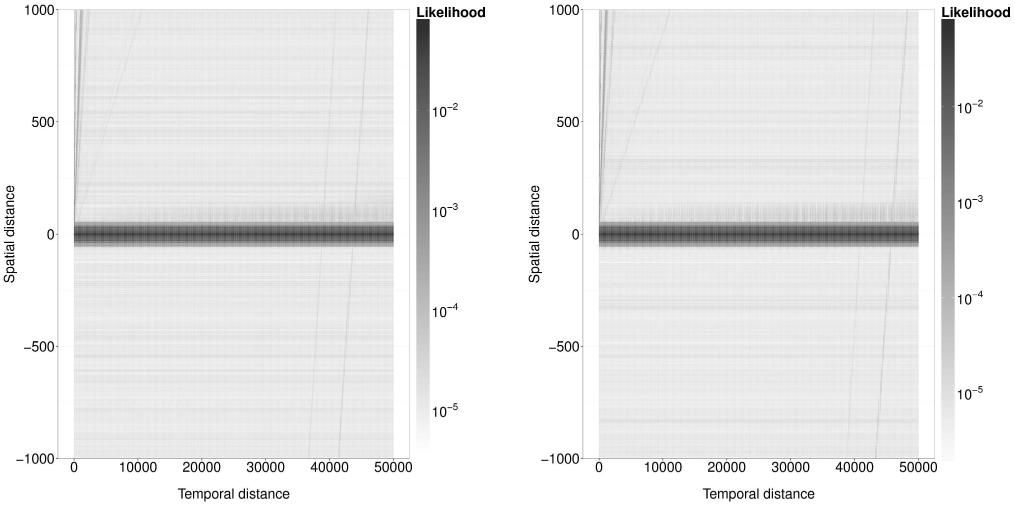
The first scatter representation – Figure 3.13a – shows a set of consecutive memory addresses accessed over time. This representation fails to capture any locality pattern. On the other hand, the second plot in Figure 3.13b, the reuse distance cumulative distributions (CDF) as defined in [134], exposes the temporal locality characteristics, but fails to quantify the spatial locality. Finally, our proposed spatio-temporal representation – Fig. 3.13c – precisely indicates that, given a current memory access, there is a significant probability to access a memory location which is $4 \cdot 10 = 40$ bytes away from the current access, in exactly 16 memory accesses later and that the probabilities of other locality patterns are significantly lower. For this latter representation, the Y-axis represents the temporal dimension (t) of the newly proposed metric (in number of consecutive accesses to memory), while the X-axis represents the spatial component (s) of the metric (in number of bytes). The intensity of a given heat-map "pixel" is proportional to the value of the $p(s,t)$ metric defined in Subsection 3.3.3.

When the heat-map shows the probability to access a memory address at a distance of exactly s bytes from the currently accessed memory address in the next exactly t memory references, we call this representation a PDF:PDF spatio-temporal locality map (probability distribution function in both spatial and temporal dimensions). Variations of this plot are possible, e.g., the PDF:CDF heat-map (probability distribution function in the spatial dimension and cumulative distribution function in the temporal dimension) would show the probability to access a memory address at a distance of exactly s bytes from the currently accessed memory address in one of the next t memory references – or the CDF:PDF plot which would show the probability to access a memory address at a distance of at most s bytes from the currently accessed memory address in the next exactly t memory references.

Next, we apply the locality heat-map concept to the Graph 500 implementation. Figure 3.14 shows examples of heat-maps for two problem sizes (scales 14 and 15) of the *seq-list* Graph 500 implementation.

The spatio-temporal memory patterns are similar across the different problem sizes. There is an area of relatively high locality between -50 and +50 bytes in the spatial dimension and low locality beyond that (Figure 3.14 shows the [-10^3,10^3] interval on the Y axis, but we actually plotted the heat-map up to [-10^5,10^5]). This means that there is a rather high probability that, given a memory access, a memory access at a distance [-50,50] in space from the current access will be accessed in the near future. Moreover, on the temporal axis, there is rather little variability for a fixed spatial distance s, which means that there is roughly the same probability to access a memory location again at any number of accesses in the future. The reuse distance chart in Figure 3.12 can be used to draw the same conclusion, because the slope of the reuse distance cumulative distribution function is approximately constant between 2^{10} and 2^{16}, but only for $s = 0$. The locality heat-map representation generalizes conclusions we can draw about the temporal characteristics to the entire s (spatial) space.

Before moving to the next PISA characterization metric, we briefly discuss the use cases of the locality heat-maps presented in this chapter. The locality method has potentially multi-fold



(a) Seq-list implementation (scale 14).

(b) Seq-list implementation (scale 15).

Figure 3.14 – Memory access patterns - Locality heat-maps.
From [22], [23].

applicability for system design. For example, when needing to reach a certain design goal, it is often the case that there is a choice to be made between either different applications to achieve that goal, different implementations of the same application or even different compiler or runtime optimizations for a given implementation. Our visualization technique allows us to compare these different ways to achieve the goal from a memory-reuse-pattern perspective and identify the optimal application, implementation or optimization.

Another application of the method lies in the prefetching domain. Our tool shows the memory access pattern of the application and, thus, helps in determining what type of prefetching would enhance performance most. Badawy et al. [30] presents an in-depth study of prefetching techniques and their relative performance on several popular algorithms. If the pattern is relatively regular hardware-prefetching techniques could be useful to address the CPU-memory performance gap. Otherwise, if the pattern is irregular, software-prefetching could be a better solution to hide memory latencies, provided that the system has enough memory bandwidth. Furthermore, the probability distribution generated by our tool can be interfaced with a prefetching engine and serve as the basis for that engine's decisions of when and what to bring from memory ahead of time.

The locality heat-maps could also be used in the context of distributed systems, where applications run on several nodes. Indeed, locality can be used to bridge the gap between fast local processing and slow remote data operations. The memory access distribution can be used to answer such questions as what the optimum amount of local memory is that would ensure a certain ratio of local-to-remote processing.

Finally, another potential use case of the locality analysis is to enable memory system design-space exploration. The locality distributions could be used in a subsequent offline processing step to estimate the performance of different memory architectures, including associativities and replacement policies. In our proposed methodology for full-system performance evaluation we will, however, use only the temporal reuse distance distribution, assuming fully associative LRU (Least Recently Used eviction policy) caches. Not only that this method estimates cache miss rates with reasonable accuracy as shown later in this chapter, but also allows us to analytically evaluate the cache performance without running simulations. We have not yet found a fast way of using the heat-map with an analytic cache performance model. One challenge is how to actually use the Markovian representation of the heat-map in combination with a probabilistic cache performance model and what assumptions to make about the independence of subsequent memory accesses. We think that even if we find a good inter-memory-address independence assumption, the analytic model would be computationally demanding, which would convert the analysis in a simulator that would reduce the desired exploration speed of our system design methodology.

3.4.4 Branch Entropy

Figure 3.15 shows the branch misprediction rate results based on the branch entropy as measured by PISA for the two Graph 500 sequential implementations for *scale* 15. PISA's estimated misprediction rate improves with increasing information on past branch behavior (given by the history size).

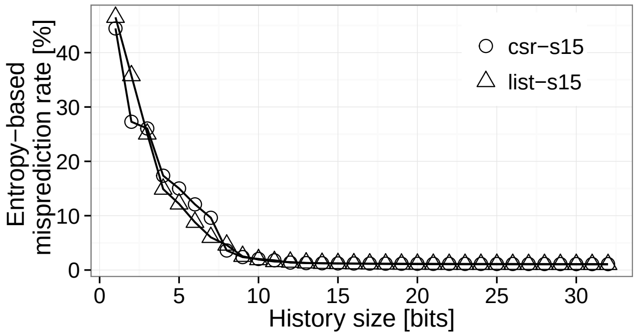


Figure 3.15 – PISA-based Graph 500 global branch entropy.
From [23] ©Springer Science+Business Media New York 2016.

As expected, for low history sizes, the branch misprediction rate is very high (just below 50%). As we assume the branch predictor to have access to more history, the misprediction rate drops steeply and stabilizes at approximately 1%. The figure also shows that increasing the history size beyond 12 only brings minor benefits. There are differences between the two Graph 500 implementations, particularly for small history sizes. However, once the misprediction rate stabilizes, the difference is very small.

Given a certain branch predictor design that we would like to evaluate, we can plug in that specific branch predictor history size in PISA and obtain an estimate for the expected branch misprediction rate from the curve. Conversely, should our goal be to find the best predictor design for an application, PISA allows us to answer questions such as: what is the best achievable misprediction rate for an application or up to what size does it make sense to increase the predictor resources to achieve that best rate.

3.5 Comparison with Real Systems

In this section we compare the application characteristics extracted by PISA with the same characteristics measured on real systems. When executing an application on a particular architecture, due to specifics related to the back-end compiler and ISA among others, the characteristics we measure will be different from those extracted by PISA to a larger or smaller extent. We perform this comparison precisely to quantify this difference. We show comparison results for the instruction mix, the L1 data cache hit rate, as estimated by the reuse distance distribution, and the branch misprediction rate, as estimated by the branch entropy, obtained with PISA vs. real profiling data obtained on two systems.

All the applications have been compiled with the `-O3 -fno-slp-vectorize -fno-vectorize` optimization flags. We have performed the measurements on two systems: (1) E5-2690 Sandy Bridge processor (64-bit architecture, 160 integer register file size, 144 floating-point register file size, issue width of 6, cache line size of 64 bytes, L1 data cache size of 32K, L1 instruction cache size of 32K, L2 cache size of 256K and L3 cache size of 20M) and (2) 8286-42A POWER8 processor (64-bit architecture, issue width of 10, cache line size of 128 bytes, L1 data cache size of 64K, L1 instruction cache size of 32K, L2 cache size of 512K and L3 cache size of 8M).

To decide whether to instrument in addition to the applications the external libraries that they use, we have profiled the execution on the POWER8 machine using *oprofile* and monitored the processor performance counters to estimate the proportion of external library instructions. The results showed that the proportion is on average 1.33% across the SPEC CPU2006 and Graph 500 *seq-list* and *seq-csr* applications (85% of which are spent in the `libc` library). We considered this percentage small enough for PISA to only analyze the application's own instructions.

3.5.1 Instruction Mix

Figure 3.16 shows how the Graph 500 instruction mix extracted with PISA compares with its corresponding instruction mix measured on an x86 Sandy Bridge and a POWER8. To profile the instruction mix, on x86, we used MICA [72], a PIN-based tool and, on POWER8, we used *operf* to monitor the hardware performance counters. The instruction mix consists of the proportion of occurrences of each instruction type (loads, stores, control, floating-point and integer) relative to the total number of instructions.

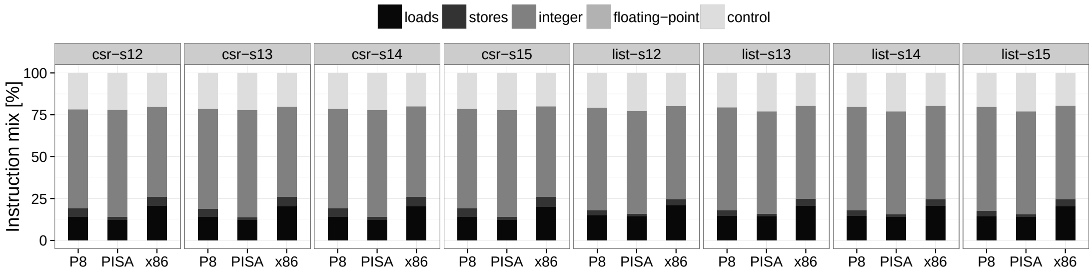


Figure 3.16 – Graph 500 instruction mix - POWER8 and x86 vs. PISA.
From [23] ©Springer Science+Business Media New York 2016.

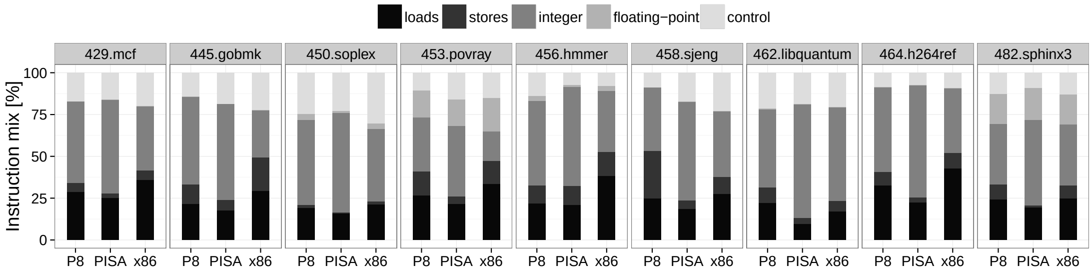


Figure 3.17 – SPEC CPU2006 instruction mix - POWER8 and x86 vs. PISA.
From [23] ©Springer Science+Business Media New York 2016.

To compare PISA's results with the real profiling measurements, we use the root mean square error (RMSE) between proportions across instruction types. For the two Graph 500 sequential implementations, PISA extracts the instruction mix with high accuracy when compared with both processor architectures. The RMSE for the different problem sizes of *seq-csr* is below 2.5 percentage points vs. P8 and below 6.2 percentage points vs. x86, while for *seq-list* it does not exceed 1.4 percentage points vs. P8 and 4.2 percentage points vs. x86. One interesting observation is that the instruction mix of these particular applications is relatively independent of the problem size and PISA is able to capture this characteristic.

The main source of error pertains to the memory instructions category which is underestimated on the PISA side. This result can be explained by the fact that our characterization assumes an ideal processor model with an unlimited number of registers. In reality, the number of registers is limited and registers can spill resulting in an increase of the number of memory instructions. Moreover, the accuracy is lower for the x86 processor than for P8. This can be explained by the fact that LLVM IR is a RISC-like instruction set, thus, closer to P8, whereas x86 uses a CISC instruction set architecture.

To understand how PISA performs on a broader set of benchmarks, we show in Figure 3.17 the instruction mix results for multiple SPEC CPU2006 benchmarks. In contrast to the Graph 500 results, here we see a higher degree of variability of the characterization accuracy. Although the average RMSE vs. P8 across the benchmarks is 7.5 percentage points, we have benchmarks for which the RMSE is as low as 3.9 percentage points (*429.mcf*) and benchmarks for which the RMSE is as high as 14.8 percentage points (*458.sjeng*). Similarly for x86, we see an average RMSE across the benchmarks of 11 percentage points, but with individual values between 6.6 (*462.libquantum*) and 15.7 (*445.gobmk*) percentage points.

Although the accuracy is lower than in the case of Graph 500, we should also note that the SPEC CPU2006 benchmarks exhibit much higher differences between different processor architectures than Graph 500 does. Indeed, whereas with Graph 500 the RMSE *between architectures* is on average 5 percentage points, the RMSE for SPEC CPU2006 is on average 11 percentage points. PISA being architecture-agnostic, it cannot possibly exhibit an error far smaller than the spread an application exhibits between architectures. Strengthening this conclusion is the fact that the applications where PISA has a lower accuracy (e.g., *458.sjeng*, *445.gobmk*) are precisely the applications that show the largest spread between architectures.

3.5.2 Level-1 Cache Hit Rate

Figure 3.18 shows how the Graph 500 L1 cache hit rate, modeled with PISA using the data reuse distribution, compares with its corresponding L1 cache hit rate measured on a POWER8 (L1 cache size of 64KB, L1 data cache line size of 128 bytes). To profile the L1 cache hit rate we used the *perf* tool to monitor the total number of load and store instruction misses. We performed the same study on x86 using the same profiling tool. As the results are similar to POWER8, for brevity we only show here the POWER8 vs. PISA comparison.

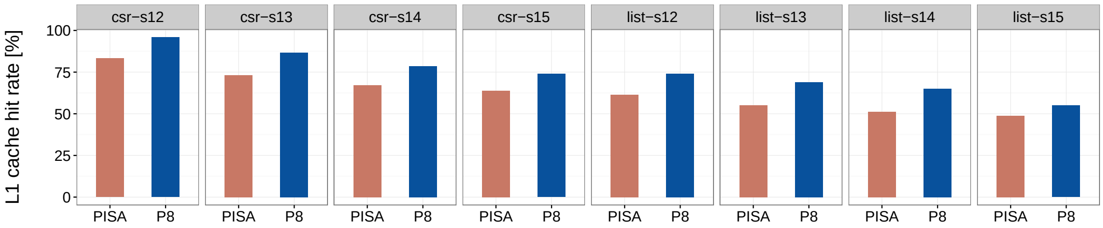


Figure 3.18 – Graph 500 L1 cache hit rate - PISA vs. POWER8.
From [23] ©Springer Science+Business Media New York 2016.

The estimated vs. measured RMSE across these Graph 500 implementations and problem sizes is 12 percentage points. While slightly high, this is a reasonable result considering that PISA is oblivious to the micro-architectural cache details and only takes into account the cache size and cache line size. Furthermore, even small variations in the hit rate of the application, e.g., as a consequence of changing the problem size, are correctly reflected in the estimations PISA provides. For example, we notice that the L1 cache hit rate of Graph 500 on POWER8 decreases with larger problem sizes for both *seq-csr* and *seq-list* and PISA is able to capture this specific characteristic of the application. Finally, when we extend the study to a broader

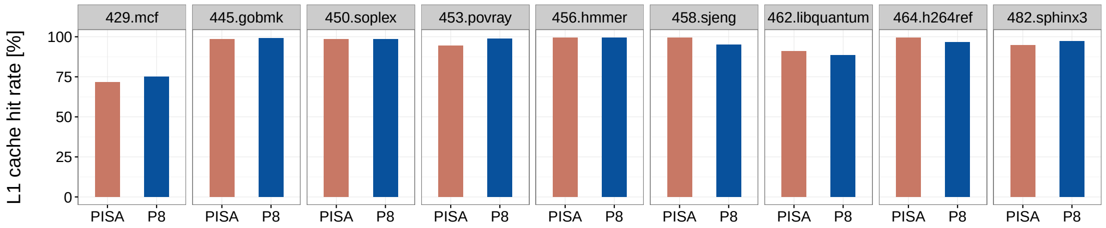


Figure 3.19 – SPEC CPU2006 L1 cache hit rate - PISA vs. POWER8.
From [23] ©Springer Science+Business Media New York 2016.

range of applications (SPEC CPU2006 – Figure 3.19) we conclude that outside of Graph 500, the precision of the PISA estimates is actually significantly higher: the RMSE across these SPEC CPU2006 benchmarks is only 2.8 percentage points.

We also analyzed the LLC hit rates, but we did not see much variability between the applications. We have measured the LLC hit rates by monitoring four perf performance events on the POWER8 machine: LLC-store-misses, LLC-load-misses, LLC-loads and LLC-stores. The hit rates obtained were between 97% (for *429.mcf*) and approx. 99% (for the others). PISA reports similar LLC hit rates, between 97% (for *429.mcf*) and 99.99%.

3.5.3 Branch Misprediction Rate

The global branch misprediction rate estimated by PISA using the branch entropy analysis is generally an optimistic estimator for the real misprediction rate [132]. Indeed, this analysis quantifies the performance of an ideal branch predictor, that is, a predictor that knows in advance, before the application is run, all the sequences of a given length of branch outcomes and their corresponding probability of occurrence.

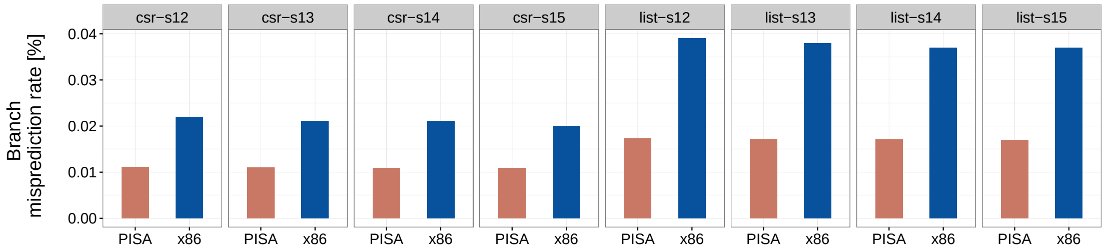


Figure 3.20 – Graph 500 branch misprediction rate - PISA vs. x86.

Figure 3.20 shows the comparison between the branch misprediction rate estimated with PISA, for a global-history size of 32, which corresponds to the size of the global buffer on an x86 Sandy Bridge processor [63], and the real measurement on x86 for Graph 500. The real measurements were obtained using the *perf* tool version 3.2 which collects information about the branch misses. The figure shows that as the problem size increases, the misprediction rate slowly decreases, e.g., for *seq-csr* the real misprediction rate decreases from 2.58% to 2.37%

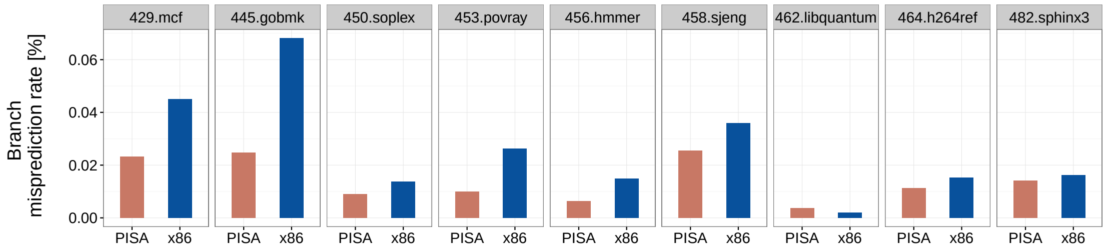


Figure 3.21 – SPEC CPU2006 branch misprediction rate - PISA vs. x86.

when increasing the problem size from scale 12 to scale 15. PISA's estimate also exhibits a decrease albeit a less pronounced one (from 1.11% to 1.09% in the same *seq-csr* case). Across implementations, the same figure shows that the *seq-list* version of Graph 500 exhibits a higher misprediction rate than the *seq-csr* version (e.g., for scale 15, *seq-csr* has a misprediction rate of 2% vs. 3.7% for *seq-list*), which is also captured by PISA, once a slightly lower amplitude (1% vs 1.7% for the same scale-15 case).

Figure 3.21 shows the same analysis results for SPEC benchmarks. Even when looking at this broader range of applications, we notice that PISA generally provides an optimistic estimator of the branch misprediction rate and is equally able to identify the best and worse performing applications from this perspective. Indeed, PISA estimates that *429.mcf*, *445.gobmk* and *458.sjeng* will experience a significantly higher misprediction rate than the rest, and, conversely, that *462.libquantum* will experience a much lower misprediction rate.

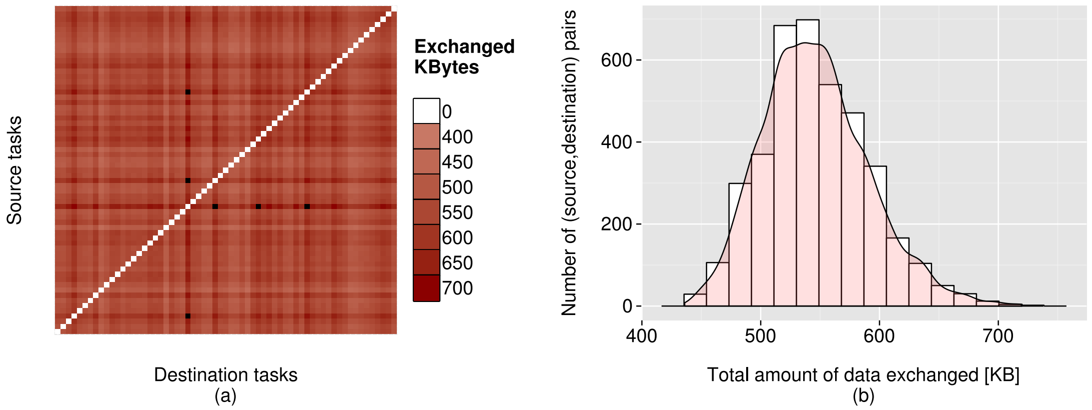
To quantify to what extent misprediction rate trends across applications and input problem sizes are captured by PISA, we trained a linear regression model having the x86/P8 misprediction rate as the response variable and the PISA estimate as the predictor variable. Across the SPEC CPU2006 benchmarks, the R-squared of this model has a value of 0.77 (x86) and 0.76 (P8), respectively, which indicates a reasonably good linear correlation. As future work we are planning to improve the branch entropy model by looking at the entropy based on local-history sizes and by considering additional metrics that characterize the branch behavior of an application.

3.5.4 Communication Patterns

Figure 3.22 shows the communication pattern extracted with PISA for a representative single BFS computation of the MPI-simple implementation of the Graph 500 benchmark, for scale 20, edge factor 64 and 64 concurrent processes.

The figure first indicates that the communication pattern of this benchmark is uniform all-to-all, as expected from the measurements we performed on the same application run on the MareNostrum supercomputing system (see Subsection 2.5).

Moreover, the same figure shows that the average total amount of data exchanged between two



(a)

(b)

Figure 3.22 – Representative single BFS computation for scale 20, edge factor 64 and 64 concurrent processes. Figure (a) illustrates the traffic matrix between the processes and suggests that data exchanges are approximately uniformly distributed between all possible (source,destination) pairs. Figure (b) shows the actual distribution of (source,destination) pairs communication volumes across possible data amounts. The volumes are distributed approximately Gaussian around a mean of 554 KB with a standard deviation of only 8.1% and a slight positive skew.

communicating MPI processes is 554 KB with a standard deviation of 8.1%. We also calculated (not shown in the figure) the average message size of the communication exchanges. For the same problem size and number of processes we obtained an average message size of 4 KB with no variability (zero standard deviation).

Finally, we compare the PISA communication pattern results with the pattern measured on the MareNostrum supercomputing system for the same problem size and number of processes (the measurements shown in Figure 2.5). The comparison shows that PISA slightly overestimates the total amount of data exchanged per pair of communicating processes by 8% and the average message size by 6.6%. This can be explained by the inherent variability of the Graph 500 of the graph generation phase and the allocation to the MPI nodes.

3.6 Related Work

Caparros [43] gives a top-level overview of a workload analysis framework based on the LLVM interpreter which is no longer supported by the compiler community. Although we share the same characterization objectives, the solution proposed in [43] is limited to single-threaded applications, and it is incompatible with basic programming-languages constructs, e.g., functions that take as argument a pointer to another function. Moreover, the interpreter does not allow access to the real memory addresses accessed by the program, thus, limiting the workload analysis opportunities.

Shao et al. [116] propose an architecture-independent analysis framework based on the IDJIT intermediate representation (IR) form as an alternative to the LLVM IR code. The proposed framework measures the application's instruction mix as well as the memory and branch entropies. Once analyzed, the high-level IDJIT bytecode is translated into LLVM IR and the LLVM compilation is triggered to generate the executable binary. The IDJIT-based solution is shown to be compatible with single-thread implementations only. To the best of our knowledge, there is no IDJIT-based solution functional for parallel OpenMP and MPI workloads. As our target is to model large-scale systems, we need to extract software properties from parallel applications, which is precisely what PISA allows us to achieve. Moreover, many available benchmarks for HPC systems are implemented in Fortran [33]. The IDJIT-based solution is, however, not compatible with Fortran code. Finally, even if an IDJIT-based solution would be available for analyzing OpenMP and MPI code, including Fortran, instead of using IDJIT as an additional IR form and processing step in the LLVM compilation process, we directly instrument and then analyze the LLVM IR code at application run-time. The advantages presented in [116] of the IDJIT IR over the LLVM IR are actually automatically handled in our proposal by the LLVM *mem2reg* optimization.

Hoste et al. [72] introduce an ATOM/Pin-based [93] workload instrumentation and analysis tool. The analysis results, however, depend on the hardware architecture and ISA on which the analysis was performed. Unlike ATOM/Pin, which instruments an already generated machine binary, we instrument and analyze the LLVM IR code, which is a higher-level code representation that does not suffer from, e.g., machine-dependent calling conventions or register spilling, as found in a machine-specific binary.

Our work can also be compared with existing research in the field of Graph 500 profiling and characterization. Previous research on this topic is limited to empirical profiling data of the MPI implementations. Jose et al. [79], Suzumura et al. [124] and Anghel et al. [21] evaluate the performance of the Graph 500 parallel implementations and report profiling results for computation and communication times. To our knowledge, our work is the first to propose a characterization of the hardware-agnostic compute properties of this benchmark, instruction-level parallelism and memory-access properties.

Regarding our proposed memory locality approach, the data locality properties of programs have been extensively studied in the context of both memory design and code optimization. Beyond qualitative descriptions provided in computer architecture books [68], different locality characterization metrics have been proposed in literature. In terms of temporal locality, Changwoo et al. [48] introduce reference distance as the total number of references between accesses to the same data. Beyls et al. [36] show that this metric cannot exactly predict cache behavior for fully associative caches, but the reuse distance or stack reuse distance [58] can. This metric is defined as the number of distinct data elements accessed between two consecutive references to the same element. We use this metric to measure with PISA the temporal memory reuse pattern of the application.

In terms of spatial locality, the previous related work has attempted to quantify it via mainly scalar metrics that allow for easy ordering or clustering of applications in locality classes. This has been done for example by using some form of reduction function to aggregate distances between consecutive or close to consecutive memory accesses [130] or by looking at locality from the perspective of the efficiency of cache line usage [101]. This related work, however, tends to treat the spatial and temporal dimensions of locality as orthogonal to each other, and, thus, only offer a pair of unidimensional, often even scalar views on the way applications handle data accesses. With our proposed locality heat-maps we generalize these concepts and quantify accurately the entire two-dimensional spatio-temporal locality characteristic of a program. This approach has the drawback of replacing a small set of values with a locality signature for each application, thus, making it difficult to, e.g., categorize applications in classes of locality patterns. However, it offers a more complete view of the application properties, allowing for increased optimization potential, either via memory system or cache design or via ahead-of-time pre-fetching strategies.

Looking at the broader context, PISA will be used as a workload-model generator for a set of analytic hardware models for performance evaluation, as shown in the next chapter. Thus, the combination of PISA with analytic models can also be compared with other performance evaluation approaches, ranging from theoretical performance models [117, 54] to system simulators and emulators, e.g., COTSon [25], MARSSx86 [105], or Graphite [35]. The latter are capable of generating high-accuracy performance estimates, incurring, however, a high evaluation overhead, while the former produce estimates rapidly, but fail to capture details of application behavior as the application is modeled abstractly. We aim to provide a middle ground between the two, by complementing purely analytic performance models with a detailed hardware-independent application model extracted by PISA. We show how PISA performs in combination with compute and communication models in Chapters 4 and 5.

3.7 Conclusions

In this chapter we presented a framework for architecture- and ISA-agnostic workload characterization using a novel instrumentation of native application code. Using the Graph 500 benchmark suite as a representative graph analytics code, we have illustrated how our framework can be leveraged to extract application properties relevant for performance evaluation, such as memory access patterns, branch behavior and instruction-level parallelism.

We showed that the framework is able to capture such properties as (1) the ILP that an application exhibits when assuming an *ideal machine model*; (2) the subsequent reduction of the exploitable ILP when integrating general hardware constraints, such as limited instruction window width or control flow restriction; 3) detailed spatio-temporal data locality patterns that generalize the traditional reuse-distance analysis; 4) the variability of the potential for instruction-level parallelism across threads or processes of parallel applications; and 5) the branch behavior of an application.

Furthermore, we compared PISA's results with real measurements on two processors using Graph 500 and SPEC CPU2006 benchmarks. The results indicate that PISA extracts the instruction mix of an application with high accuracy for RISC-like ISAs such as the POWER8 ISA and with reasonable accuracy for x86. Moreover, PISA's data reuse distribution estimates with good accuracy the L1 cache hit rate for the SPEC CPU2006 benchmarks when compared with both x86 and POWER8 processors. Furthermore, PISA generally provides an optimistic estimate for the branch misprediction rate and its branch-entropy-based predictions exhibit a high correlation with real measurements across applications.

We believe that such a framework is a key tool in enabling efficient and high-quality application characterization as well as easy comparison of the suitability of different systems for arbitrary workloads. In the next chapter, we will describe different applications of PISA to processor performance modeling.

4 Analytic Processor Modeling Using Hardware-Agnostic Software Profiles

In this chapter we present how hardware-agnostic software profiles, such as PISA's, can enable processor performance modeling. We show how software properties such as instruction mix, instruction-level parallelism, data and instruction memory reuse patterns can be used with analytic models to estimate the processor core performance. We load the software properties extracted with PISA into two state-of-the-art analytic approaches and we validate the estimates with performance measurements performed on real systems. To the best of our knowledge, we contribute with the first analysis of the accuracy of the combination of analytic processor performance models with hardware- and ISA-agnostic software profiles.

We also analyze in detail the modeling of one processor core component, the branch predictor. We use PISA to extract a trace of branch decisions during the execution of a program. This trace is used as input to two methods that characterize the predictability of the branch behavior of an application. These are the branch entropy [132] and a novel max-outcome branch prediction method. Our contributions related to branch misprediction modeling are the following. (1) We investigate how accurately the branch entropy models hardware branch predictor parameters and describe a method of reverse engineering the global history size of a branch predictor. (2) We analyze the limitations of branch entropy and propose a method to derive analytic models of hardware branch prediction. (3) We propose a novel method for the characterization of application branch behavior which is not only correlated with the measured branch miss rates, but it is also more accurate than the branch entropy.

4.1 Introduction

Simulations are often used by researchers and designers to study the performance of processors. Although they are usually reasonably accurate in predictions, simulations are slow, especially in the context of large-scale design-space exploration. A single simulation provides little or no insight into the main hardware-software interactions that occur in a processor. A large set of slow simulations is normally required to identify trends and dependencies between the different architectural aspects that impact the performance. In this work, we use PISA as an enabler of fast processor performance prediction. In the first part of this chapter, we provide an overview of how the PISA software properties can be used with existing analytic performance and power models. These results are relevant for system designers and researchers to understand the feasibility of analytic hardware modeling using hardware-agnostic profiles. In the second part of the chapter, we focus in detail on the modeling of one processor component, namely the branch predictor.

For the remainder of the thesis, we assume models of superscalar out-of-order processors. The instructions are fetched from the instruction cache, decoded and dispatched to the issue queue and the reorder buffer. The issue queue is used for out-of-order scheduling and the re-order buffer is used for in-order committing of the instructions. Instructions with all data dependencies met are ready for execution and dispatched to a functional unit of type corresponding to the type of the instructions. For example, the load/store instructions are dispatched to the load/store units or the floating-point instructions are dispatched to the floating-point units. Moreover, each processor has private L1 and L2 caches, while the L3 cache is shared across multiple cores. We also assume that the processor is attached to a main memory DRAM.

For in-order processors, such as the PowerPC A2 of Blue Gene supercomputers, we use the same models slightly adapted to the in-order core architecture. For example, on the software side, we measure from the instruction flow an in-order ILP instead of out-of-order ILP. Also, on the hardware side, the reorder buffer does not exist in in-order processors, thus, we remove the performance models or constraints related to this processor buffer.

The structure of this chapter is the following. In Sections 4.2 and 4.3 we analyze the accuracy of using platform-agnostic software profiles with analytic processor performance and power models. We continue in Section 4.4 with an in-depth study on how to analytically model branch prediction in a processor. We present related work in Section 4.5 and conclude in Section 4.6.

4.2 Processor Performance Modeling

The processor core performance can be expressed in instructions per second, d or in cycles per instruction (CPI), c. One can be converted to the other via the clock operating frequency f_{core}: $d = \frac{f_{\text{core}}}{c}$. The CPI performance is impacted by multiple machine events that may occur during program execution, such as instruction executions, data or instruction cache misses or branch mispredictions.

Stanley-Marbell [123] proposes a mechanistic performance model for superscalar out-of-order processors that quantifies the CPI penalty effect of these machine events independently. Jongerius et al. [77, 76] present an alternative processor performance model that also includes the effect on performance of the interaction of the different machine events. Both models in [123] and [77, 76] are inspired from a mechanistic model presented in [59]. In this section, we briefly explain how hardware-agnostic application properties can be used with the two compute models presented in [123] and [77, 76].

4.2.1 Overview of Independent Modeling of CPU Events

The processor model in this chapter is based on the model presented in [123]. The machine events covered by [123] are instruction executions, instruction stalls, data stalls and branch mispredictions. We will show in this chapter how to use PISA profiles to model the performance of these events. It is worth mentioning that [123] also models memory bandwidth limitations that we do not cover in this chapter. In the following, c^i will refer to the penalty in cycles that every event incurs on average.

The instruction executions are determined from the application's scalar instruction mix and average ILP per instruction type against the core architectural issue-width and functional-unit count. The issue-width is the maximum number of instructions that can be issued during the same cycle. The instruction mix and the average ILP per instruction are calculated with PISA (see Subsection 3.3).

The ILP that can be exploited for each instruction type depends on both the inherent application ILP and the number of available execution units. Thus, the exploitable ILP can be expressed as $\hat{n}^{\text{type}} = \min_{\text{type}}(n^{\text{type}}, \text{ILP}^{\text{type}})$, where n^{type} is the number of functional units for a given instruction type and ILP^{type} is the ILP of the application for that particular type of instructions. The *type* can be either integer (INT), floating-point (FP), control (CTRL), or memory (MEM).

To determine the exploited ILP we use an additional model to the ILP model in [123]. If the number of available functional units for a given type n^{type} bounds the exploited ILP for that type, the overall application performance will degrade due to resource contention. The overall ILP exploited by the application is bounded in this case by the maximum slowdown of any individual type, $S = \max_{\text{type}}(\frac{\text{ILP}^{\text{type}}}{\hat{n}^{\text{type}}})$. Additionally, the overall ILP cannot be larger than the core issue width. Thus, the overall ILP can be modeled as $\hat{n} = \min\left(n^{\text{issue-width}}, \sum_{\text{type}} \frac{\text{ILP}^{\text{type}}}{S}\right)$. In summary, the time penalty incurred by the instruction execution events $c^{\text{executions}}$ is:

$$c^{\text{executions}} = \frac{1}{\hat{n}}. \tag{4.1}$$

The next machine events that we model with PISA profiles are the data stalls. The performance of a processor is impacted by the cycles required to complete the memory access instructions. Indeed, memory access latencies and finite bandwidths introduce stalls in the execution of the instruction stream. We model the data stalls by considering fully-associative caches with least-recently used (LRU) eviction policy and by ignoring hardware prefetchers.

We use PISA to extract the temporal data reuse distribution of an application (see Subsection 3.3.3). The spatial-temporal locality heat-maps presented in the same Subsection 3.3.3

would be a more complete characterization of the application memory access pattern. However, our initial cache performance models that use the heat-map as input were too slow for the desired capacity of our full-system performance prediction methodology. Therefore, for the data stalls, we only use the temporal data reuse distribution. However, as shown in Subsection 3.5, this property allows modeling the cache miss rate with reasonable accuracy.

To model the cache misses we use the application data reuse distribution with hardware parameters, such as the cache size and the cache line size. More precisely, the fraction of memory operations that request cache lines with a reuse distance larger than the number of lines that fit the cache, is assumed to incur a cache miss event. Indeed the cache size is smaller than the cache line size multiplied with the reuse distance of that cache line. We formalize this model for the L1 data-cache as shown in Equation 4.2:

$$c^{\text{L1d-stalls}} = T^{\text{core-L2}} \cdot F^{\text{MEM}} \cdot f^{\text{miss}}(D^{\text{d-reuse}}, M^{\text{d-L1}}, L^{\text{d-L1gran}}), \tag{4.2}$$

where $f^{\text{miss}}()$ calculates the fraction of misses from the data reuse distribution $D^{\text{d-reuse}}$ for an L1 data cache size of $M^{\text{d-L1}}$ and a cache line size of $L^{\text{d-L1gran}}$. F^{MEM} is the fraction of instructions that perform a memory operation. Thus, $F^{\text{MEM}} \cdot f^{\text{miss}}(D^{\text{d-reuse}}, M^{\text{d-L1}}, L^{\text{d-L1gran}})$ represents the fraction of instructions that result in an L1 data cache miss. To quantify the time penalty in cycles for an L1 data cache miss, we multiply the cache misses with the core-to-L2-cache latency, $T^{\text{core-L2}}$.

Similarly, we calculate the data cache miss rates for the L2 and L3 caches. For the L3 cache misses, the DRAM access latency is modeled similarly to [59]. Instruction dispatch will not stop immediately when an L3 cache miss occurs, but will continue until the reorder buffer is filled. The DRAM latency is reduced by the latency to fill the reorder buffer which is modeled as the length of the reorder buffer divided by the overall exploited ILP $\hat{n}$. In summary, the time penalty incurred by the data stalls on the core performance is modeled as:

$$c^{\text{d-stalls}} = c^{\text{L1d-stalls}} + c^{\text{L2d-stalls}} + c^{\text{L3d-stalls}}. \tag{4.3}$$

Memory instructions that hit the L1 cache will also incur a time penalty that is related to the time required to access L1. This effect is modeled similarly to data cache misses as:

$$c^{\text{d-hits}} = T^{\text{core-L1}} \cdot F^{\text{MEM}} \cdot \left(1 - f^{\text{miss}}(D^{\text{d-reuse}}, M^{\text{d-L1}}, L^{\text{d-L1gran}})\right). \tag{4.4}$$

The next machine events that we model are the instruction stalls. We consider two effects: instruction cache misses and branch misprediction penalties. The instruction misses are calculated similarly to the data cache misses. Indeed, PISA extracts not only a temporal data reuse distribution, but also a temporal reuse distribution of the instructions. Equation 4.5 shows the model for the instruction cache misses, where $D^{\text{i-reuse}}$ is the application's instruction-cache reuse distribution and $T^{\text{core-L2}}$ is the time penalty in cycles to access the L2 instruction cache (usually the same as the latency of accessing the L2 data cache).

$$c^{\text{L1i-stalls}} = T^{\text{core-L2}} \cdot f^{\text{miss}}(D^{\text{i-reuse}}, M^{\text{i-L1}}, L^{\text{i-L1gran}}). \tag{4.5}$$

The branch misprediction penalty is modeled as suggested in [60], as the front-end pipeline refill time. This penalty is the time valid instructions take to enter the instruction window after a branch misprediction. For this model we set the misprediction rate to a typical value of the tournament predictors of 3% [68], the same for all applications. Given the fraction of control instructions in the application's code F^{CTRL} and the front-end pipeline depth $n^{\text{front-pipe}}$, the branch misprediction penalty is modeled as:

$$c^{\text{branch}} = F^{\text{CTRL}} \cdot n^{\text{front-pipe}} \cdot F^{\text{misprediction}}. \tag{4.6}$$

The summation of the penalties of the individual machine events above represents the base core performance (CPI) model.

$$c = c^{\text{executions}} + c^{\text{d-stalls}} + c^{\text{d-hits}} + c^{\text{L1i-stalls}} + c^{\text{branch}} \tag{4.7}$$

4.2.2 Independent Modeling: Single-Core Performance Results

This section presents performance results for a subset of the SPEC CPU2006 [122] and Graph 500 [4] benchmarks. We analyzed the applications as presented in Chapter 3, using PISA. For characterization, we used LLVM version 3.4 and the compiler optimization of `-O3`. From the SPEC CPU2006 benchmark set, we selected C and C++ benchmarks and run them with test data sets. For Graph 500, we executed the sequential list-based (SEQ-LIST) and the sequential compressed-sparse-row (SEQ-CSR) implementations with four different workload sizes, namely for scale values 12, 13, 14 and 15 and edge factor 16. An overview of the benchmarks and their input data sets is listed in Table 4.1.

The analyzed workload sizes are limited by the additional run-time overhead incurred by the PISA instrumentation and, therefore, are typically small with execution times that do not exceed 10 seconds when run without PISA instrumentation. On the other hand, we had to characterize each application only once as the application-specific parameters generated by the analysis are architecture-independent. To increase the accuracy of the performance results, instead of running PISA with no hardware restrictions for the ILP analysis and with byte memory access granularity for the data reuse distribution, we fixed the instruction window size for the ILP calculation to 54 and the cache line size to 64, values of the corresponding parameters of the processors used for measurement.

We validate the core performance estimated with the previously described analytic model against application performance measured on an actual hardware platform, an Intel®Xeon®E5-2697 v3 Haswell-EP. The values of the hardware parameters used in the analytic models are

Benchmark	**Domain**	**Input data set**
SPEC CPU2006		
429.MCF	Combinatorial optimization	test
445.GOBMK	Artificial intelligence	test
450.SOPLEX	Linear programming	test
453.POVRAY	Ray-tracing	test
456.HMMER	DNA pattern search	test (-num 10000)
458.SJENG	Artificial intelligence	test (8 - 9)
462.LIBQUANTUM	Physics (quantum computing)	144
464.H264REF	Video compression	test (5 frames)
482.SPHINX3	Speech recognition	test
Graph 500		
SEQ-LIST	Graph analytics (BFS)	s12-15, e16
SEQ-CSR	Graph analytics (BFS)	s12-15, e16

Table 4.1 – SPEC CPU2006 and Graph 500 applications used in the experiments.

shown in Table 4.2. To measure the performance (the number of cycles and the number of instructions) on the Xeon processor we used perf, version 3.2.

Figures 4.1 and 4.2 show the comparison between the PISA-based performance estimates obtained with the analytic processor model and measurements on the Xeon processor for the CPI metric and for the execution time (in cycles), respectively. The execution times are obtained by multiplying the CPI with the instruction count. In the case of the model, the instruction count is the total number of instructions measured with PISA, thus, LLVM IR instructions, whereas, in the case of the measurement, the instruction count is the total number of assembly instructions measured with perf.

For the CPI metric, we estimate performance with a mean absolute percentage error (MAPE) of 126%, whereas for the time we estimate with a MAPE of 45%. We also analyzed the correlation between the model estimates and the measurements across the applications. We obtained a correlation factor of 0.95 for CPI and of 0.84 for execution time (in cycles). These results indicate that the model, even though it has a rather large error rate on the CPI metric, preserves with reasonable accuracy the relative differences in performance across applications.

The errors of the analytic model can be attributed to multiple sources. First, some machine events such as the integer and floating-point unit stalls are not modeled. The load/store unit stalls are underestimated. The processor model assumes that each functional unit has unit latency and that its results are immediately available to the next instruction. In practice, operations can take several cycles, increasing the average CPI due to dependency stalls. For the load/store unit, the model partially models this effect by calculating the delay due to L1 accesses. However, the data cache misses are overestimated, because the model assumes that load/store operations that cause a cache miss fully serialize, while in practice independent memory operations may overlap, effectively reducing the delay. There are sources of error also

Parameter	Description	Xeon E5-2697
n_{cores}	Cores per socket	14
f_{core}	Core clock frequency	2.6 GHz
$n^{\text{issue-width}}$	Issue width	8
n^{INT}	# integer units	4
n^{FP}	# floating-point units	2
n^{MEM}	# load/store units	2
n^{CTRL}	# branch units	2
$n^{\text{front-pipe}}$	Front-end pipeline depth	7
n^{ROB}	Reorder buffer capacity	192
$M^{\text{d-L1}}$	Data L1 cache size	32 KB
$M^{\text{d-L2}}$	Data L2 cache size	256 KB
$M^{\text{d-L3}}$	Data L3 cache size	32 MB
$M^{\text{i-L1}}$	Instruction L1 cache size	32 KB
$L^{\text{d-L1gran}}$	Data L1 cache line size	64 bytes
$L^{\text{d-L2gran}}$	Data L2 cache line size	64 bytes
$L^{\text{d-L3gran}}$	Data L3 cache line size	64 bytes
$L^{\text{i-L1gran}}$	Instruction L1 cache line size	64 bytes
M^{DRAM}	DRAM size per socket	32 GB
$T^{\text{core-L1}}$	Data L1 access latency	4 cycles
$T^{\text{core-L2}}$	Data L2 access latency	12 cycles
$T^{\text{core-L3}}$	Data L3 access latency	36 cycles
$T^{\text{core-DRAM}}$	DRAM access latency	217 cycles
B^{DRAM}	DRAM bandwidth	59.7 GB/s

Table 4.2 – Values for hardware parameters of Intel®Xeon®E5-2697 v3 Haswell-EP [74].

on the software characterization side. For instance, the instruction mix as already shown in Chapter 3 can be different from the actual instruction mix run on the hardware.

4.2.3 Overview of Modeling of CPU Events and Event Interactions

Jongerius et al. [77, 76] propose an analytic model for processor-core performance that captures the interactions between machine events. In contrast to the previous model, which assumes the different stall components to be independent of each other, the model proposed in [77, 76] also considers the interactions between event types where a delay in one place causes additional contention in another. More precisely, the approach models the effect on performance of full pipeline stalls. For example, whenever a long-penalty instruction occurs, such as an L3 cache access, the issue queue will fill with instructions that have dependencies on the long-penalty instruction.

The performance is expressed, as in the previous processor modeling case, in cycles per instruction (CPI, c) or instructions per cycle (IPC, c^{-1}). The model accounts for different constraints related to the software-inherent bottlenecks and the hardware resources available

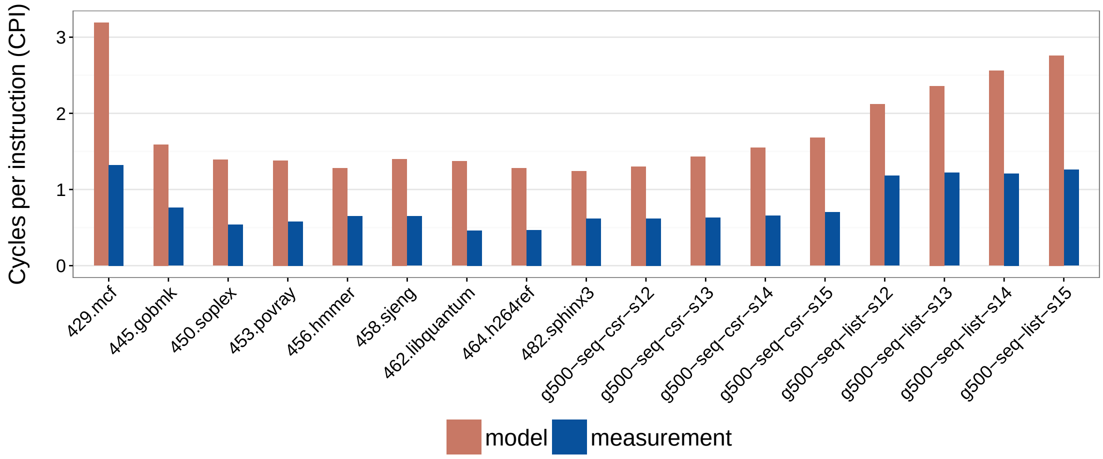


Figure 4.1 – Single-core CPI performance results: model vs. measurements on Xeon.

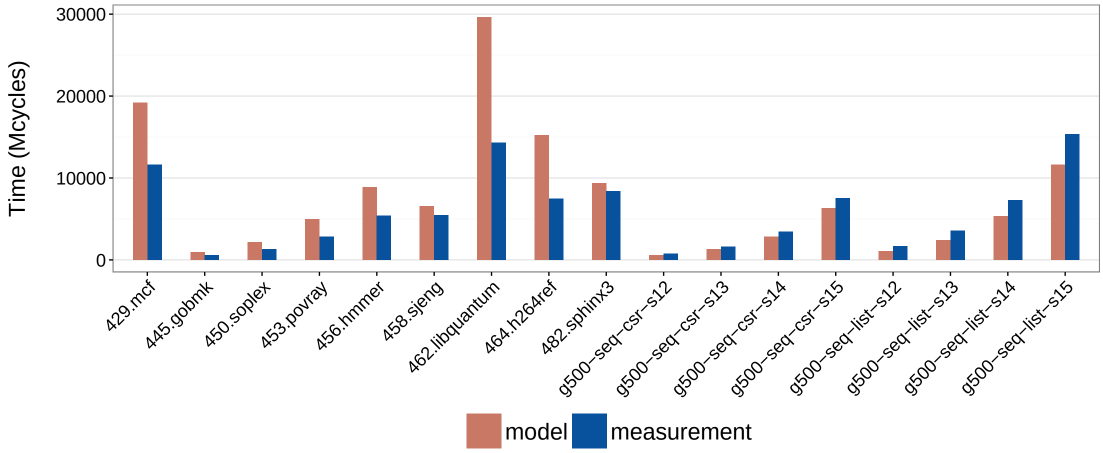


Figure 4.2 – Single-core execution times: model vs. real measurements on Xeon.

at the core level. All the constraints are solved through a linear-programming solver that finds the maximum attainable core performance. A detailed explanation of how the PISA software properties are used with all the model constraints is out of the scope of this chapter. However, we present a couple of examples of how the model uses part of the software properties.

We measure the fraction of instructions of each type F^{type} executed by the application using PISA, where *type* can be control (CTRL), memory (MEM), integer (INT) and floating-point (FP). The issue rate of instructions is limited by the number of functional units n^{type} of that *type* provided by the hardware architecture. In this case, a first set of constraints is defined as $c^{-1} \cdot F^{\text{type}} \leq n^{\text{type}}$ for each type of instructions.

Another constraint is imposed by the inherent ILP of the application. Indeed, the number of instructions in flight at the same time cannot be larger than the overall ILP of the application.

This can be formalized as follows: $c^{-1} \cdot \sum_{\text{event}} F^{\text{event}} \cdot T^{\text{event}} \leq \text{ILP}$, where $c^{-1} \cdot \sum_{\text{event}} F^{\text{event}} \cdot T^{\text{event}}$ represents the average number of events occurring at the same time and T^{event} the latency of an *event*. An event represents a cache access (L1, L2, L3), a DRAM access, an integer multiplication/division, or a floating-point multiplication/division. For the fraction of events F^{event} per instruction for the integer and floating-point multiplications and division events, we use the instruction mix extracted with PISA. We also use PISA to characterize the overall ILP of the application. The fractions of cache misses at different levels of the memory hierarchy are determined based on the temporal reuse distance distributions as exemplified in Section 3.4.3 and Subsection 4.2.1. The reuse distribution paired with the hardware parameters such as the cache size gives the hit rate on a particular cache hierarchy (assuming fully-associative caches with LRU eviction policy and no hardware prefetchers).

The same constraint can actually be used to model bounds derived from the application-inherent ILP per type, namely $c^{-1} \cdot \sum_{\text{event}_{\text{type}}} F^{\text{event}_{\text{type}}} \cdot T^{\text{event}_{\text{type}}} \leq \text{ILP}^{\text{type}}$. This captures the constraint on ILP between instructions of the same type, while the previous model captures the constraint on ILP between instructions of different types. Jongerius et al. [77, 76] propose more such constraints related to full pipeline stalls, memory bandwidth contention and branch mispredictions. The branch misprediction constraint uses application-specific misprediction estimates based either on branch entropy or other metrics as shown in Section 4.4.

4.2.4 Event-Interaction Modeling: Single-Core Performance Results

This section presents performance results for a subset of the SPEC CPU2006 [122] and Graph 500 [4] benchmarks as shown in Table 4.1. We first analyzed the applications as presented in Chapter 3, using our LLVM-based characterization framework PISA. For characterization, we used LLVM version 3.4 and the compiler optimization of `-O3 -fno-slp-vectorize -fno-vectorize`.

We validate the core performance estimated with the model in [77, 76] against application performance measured on an actual hardware platform, an Intel®Xeon®E5-2697 v3 Haswell-EP. The values of the hardware parameters used in the analytic models are shown in Table 4.2. To measure the performance (the number of cycles and the number of instructions) on the Xeon processor we used perf, version 3.2.

Figures 4.3 and 4.4 show the comparison between the PISA-based performance estimates obtained with the analytic processor model and measurements on the Xeon processor for the CPI metric and for the execution time (in cycles), respectively. The execution times are obtained by multiplying the CPI with the instruction count. In the case of the model, the instruction count is the total number of instructions measured with PISA, thus, LLVM IR instructions, whereas, in the case of the measurement, the instruction count is the total number of assembly instructions measured with perf.

For the CPI metric, we estimate performance with a MAPE of 24%, whereas for the time we

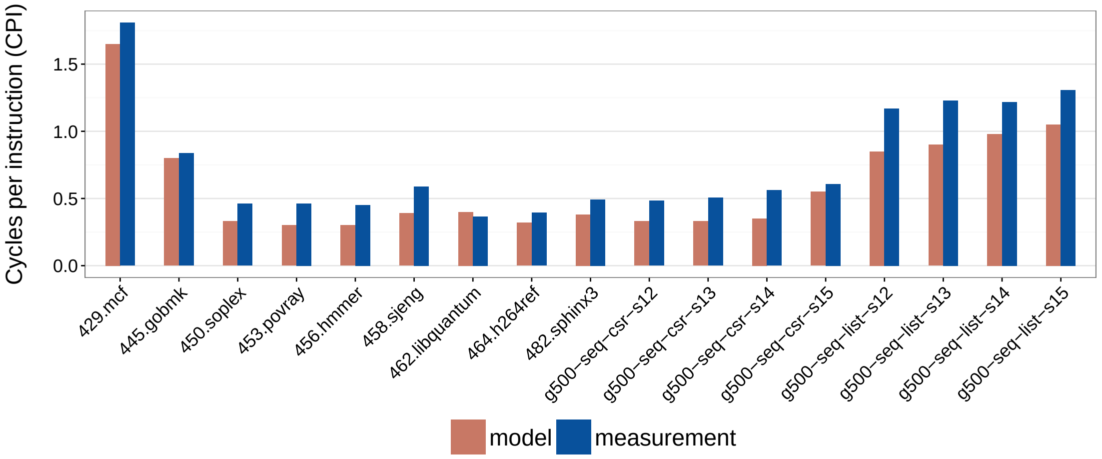


Figure 4.3 – Single-core CPI performance results: model vs. measurements on Xeon.

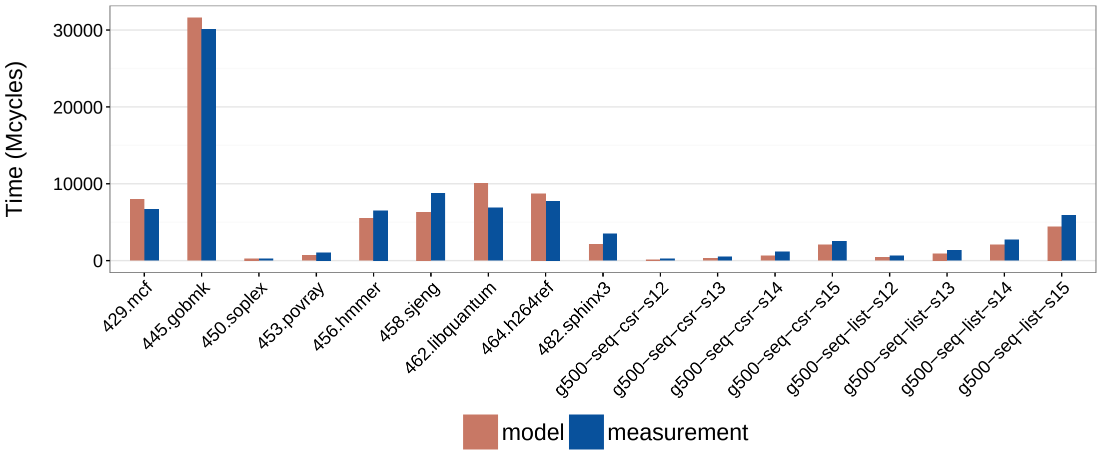


Figure 4.4 – Single-core execution times: model vs. real measurements on Xeon.

estimate with a MAPE of 34%. We also analyzed the correlation between the model estimates and the measurements across the applications. We obtained a correlation factor of 0.96 for CPI and a of 0.97 for execution time (in cycles). These results indicate that the model not only provides good accuracy, but also accurately preserves the relative differences in performance across applications. When compared with the previous compute model approach, the current model performs better in terms of both absolute and relative time predictions (relative across applications). Thus, for the remainder of the thesis, we will use this model [77, 76] with PISA profiles for evaluating the performance of a full system.

4.3 Processor and DRAM Power Modeling

The performance models in the previous section capture the dependency of the application execution time on the hardware properties. We showed how accurately we can use PISA profiles with compute performance models. In this section we focus on processor and memory power modeling. To model the processor power consumption we leverage existing tools, McPAT [91] for processor power and CACTI [126] or MeSAP [108] for main memory (DRAM) power.

McPAT is an integrated power, area and timing modeling framework that supports design-space exploration for multi-core processor configurations ranging from 90 nm to 22 nm. The framework implements a power model of microprocessors, from their micro-architectures and on-chip interconnect networks, to the influence of CMOS technology trends. The tool includes core power-consumption (including gate leakage, runtime dynamic power), caches and peripheral circuitry on core and die levels. The input to McPAT is: (1) application properties, which can be extracted from the PISA software profile, and (2) information about the processor performance that can be analytically calculated with the compute models introduced in the previous section. McPAT does not provide power estimates for DRAM memories.

For DRAM power consumption modeling, we use CACTI or MeSAP. CACTI is an integrated cache and memory access time, cycle time, area, leakage, and dynamic power model. By integrating all these models together, CACTI users can perform trade-off analysis between time, power, and area. MeSAP [108] is an analytic memory power model also for DRAM memories. MeSAP provides more accurate predictions than CACTI. Both CACTI and MeSAP can be used with PISA profiles and processor performance models to estimate the power consumption of DRAM memories.

4.3.1 Processor Power McPAT Modeling Overview

McPAT is configured using XML-files that include information about both the hardware parameters and the application properties. Examples of such hardware parameters are shown in Table 4.2 (e.g., number of functional units per type, number of cores, number of hardware threads, technology node, clock frequency rate, issue width, reorder buffer size). Examples of software-related properties are the total number of instructions per type, the total number of branch mispredictions and the total number of execution cycles. The latter is derived from the processor core performance model used with PISA.

McPAT returns power numbers for the core with private caches P^{core}, the L3 data cache $P^{\text{L3-memory}}$ and the network on-chip $P^{\text{on-chip-network}}$. Given these power estimates, we model the processor power consumption as shown in Equation 4.8.

$$P^{\text{processor}} = n^{\text{cores}} \cdot P^{\text{core}} + P^{\text{L3-memory}} + P^{\text{on-chip-memory}} + P^{\text{glu}}, \tag{4.8}$$

where n^{cores} is the number of processor cores, P^{core} includes not only the core power, but also that of the private L1 and L2 caches and P^{glu} represents the power consumption of the peripheral circuits at the processor level.

4.3.2 DRAM Power CACTI Modeling Overview

Like McPAT, CACTI uses as input information about both the hardware and the software properties. The static DRAM power is calculated as the number of dies (the total memory size divided by the capacity of a DRAM memory) multiplied with the number of memory banks and the leakage power per bank. These are all hardware parameters. The dynamic DRAM power, however, depends also on software properties. Indeed, the dynamic power is calculated as the number of reads per second multiplied with the energy per read operation plus the number of writes per second multiplied with the energy per write operation. While the energy per read/write operation are hardware parameters, the number of read/write operations per second is calculated using the load/store instruction mix, the total number of instructions extracted with PISA and the compute throughput derived from the processor performance model (Equation 4.7).

The number of DRAM reads per second is modeled as:

$$\frac{\text{Loads}}{\text{Loads}+\text{Stores}} \cdot \frac{f_{\text{core}}}{c} \cdot F^{\text{MEM}} \cdot f^{\text{miss}}(D^{\text{d-reuse}}, M^{\text{d-L3}}, L^{\text{d-L3gran}}), \tag{4.9}$$

where $F^{\text{MEM}} \cdot f^{\text{miss}}(D^{\text{d-reuse}}, M^{\text{d-L3}}, L^{\text{d-L3gran}}$ is the fraction of memory instructions that miss the L3 cache. This is derived from the temporal reuse distance distribution as explained for Equation 4.2. The number of DRAM writes is modeled similarly as:

$$\frac{\text{Stores}}{\text{Loads}+\text{Stores}} \cdot \frac{f_{\text{core}}}{c} \cdot F^{\text{MEM}} \cdot f^{\text{miss}}(D^{\text{d-reuse}}, M^{\text{d-L3}}, L^{\text{d-L3gran}}). \tag{4.10}$$

4.3.3 DRAM Power MeSAP Modeling Overview

Poddar et al. [108] proposes a DRAM model as a more accurate alternative to CACTI. Indeed, it is shown that MeSAP achieves a 26% error rate for the power consumption of the STREAM benchmark on a DDR4 memory, whereas the popular CACTI achieves an error rate of 165%. Moreover, MeSAP is two orders of magnitude faster than trace-based approaches and has a MAPE of approximately 20% for SPEC CPU2006 and Graph 500 benchmarks when used with the performance model in Subsection 4.2.3.

Like McPAT and CACTI, the MeSAP analytic model takes as input both hardware and software properties. Examples of hardware properties are the core clock frequency, the number of DIMMs in a system, the number of ranks in a DIMM, the size of a rank, the number of memory chips per rank and parameters specific of the memory chip, such as the refresh cycle, the memory type and size or the operating currents. On the software side, the properties are the number of bytes read and written from and to DRAM and the number of execution cycles of the application. The latter is actually the result of using the software profiles with hardware performance models such as those presented in the previous section.

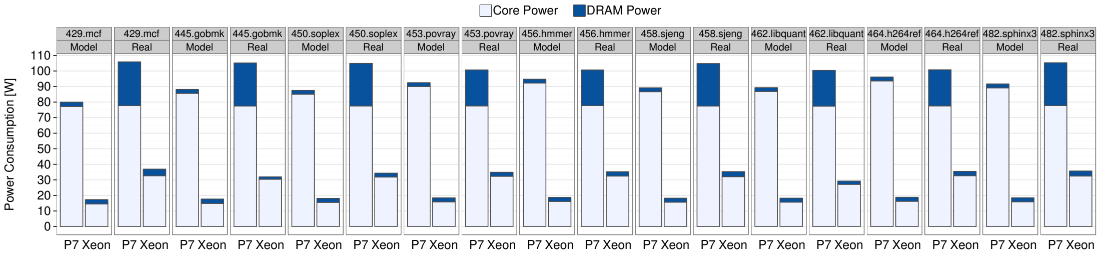
4.3.4 McPAT-CACTI Modeling: Single-Core Power Results

We present power validation results for McPAT+CACTI tools using the performance model presented in Subsection 4.2.1. We measured the power consumption on two architectures: an IBM PowerLinux-7R2 (POWER7+) and an Intel®Xeon®E5-2697 v2. On the POWER7+ system, we used the AMESTER tool [90] to measure power consumption of both the CPU package and the DRAM. The software allowed us to remotely collect power data at 1 ms resolution for core and DRAM memory sensors. On the Xeon system, we use the RAPL (running average power limit) feature and read out the model-specific registers (MSRs) to measure power consumption. Power saving and Intel TurboBoost technologies where switched off during measurements. Table 4.3 gives the hardware parameters passed to McPAT.

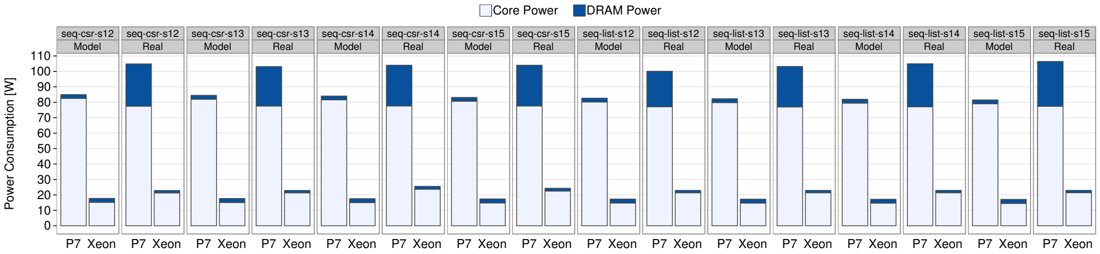
Parameter	**Description**	**POWER7+**	**Xeon**
L_{node}	Technology node	32 nm	22 nm
n_{cores}	Cores per socket	8	12
n_{threads}	Number of hardware threads	4	2
n_{sockets}	Sockets per card	2	2
f_{core}	Core clock frequency	3.6 GHz	2.7 GHz
n^{INT}	# integer units	4	3
n^{FP}	# floating-point units	4	2
n^{MEM}	# load/store units	2	2
n^{CTRL}	# branch units	1	1
$n^{\text{front-pipe}}$	Front-end pipeline depth	9	7
n^{ROB}	Reorder buffer capacity	120	168
$M^{\text{d-L1}}$	L1 cache size	32 KB	32 KB
$M^{\text{d-L2}}$	L2 cache size	256 KB	256 KB
$M^{\text{d-L3}}$	L3 cache size	80 MB	30 MB
M^{DRAM}	DRAM size	64 GB	64 GB
$T^{\text{core-L1}}$	Data L1 access latency	2 cycles	4 cycles
$T^{\text{core-L2}}$	Data L2 access latency	8 cycles	12 cycles
$T^{\text{core-L3}}$	Data L3 access latency	17 cycles	18 cycles
$T^{\text{core-DRAM}}$	DRAM access latency	375 cycles	187 cycles
$B^{\text{L1-L2}}$	L1-L2 memory bandwidth	256 GBs	80 GBs
$B^{\text{L2-L3}}$	L2-L3 memory bandwidth	512 GBs	80 GBs
$B^{\text{L3-DRAM}}$	L3-DRAM memory bandwidth	100 GBs	59.7 GBs

Table 4.3 – Power-model parameters, representing an IBM PowerLinux-7R2 (POWER7+) system and an Intel®Xeon®E5-2697 v2 (Ivy Bridge-EP). (The POWER7+ L3 bandwidth and latency values are for the non-local adaptive victim L3 [135, 73].)

Results are shown in Figure 4.5a for SPEC CPU2006 and Figure 4.5b for Graph 500. Note that



(a) SPEC CPU2006



(b) Graph 500

Figure 4.5 – Power models validation: model estimates versus real measurements of IBM PowerLinux-7R2 and Xeon-E5.

the IBM POWER7+ is an earlier generation than the Intel®Xeon®and the two follow very different design targets with respect to energy efficiency and performance. The Xeon system uses a newer technology node and is highly optimized for low energy consumption, while the POWER7+ system uses an older technology node and is optimized for performance. The POWER7+ has a larger L3 cache and employs buffer chips between memory controller and memory DIMMs to increase the size of addressable memory.

For SPEC CPU2006 we estimate power consumption with an average accuracy of 87% for POWER7+ and 53% for Xeon. The Graph 500 benchmarks show similar results, with an average accuracy for POWER7+ of 80% and 74% for Xeon. We make two observations in the POWER7+ results. (1) DRAM power is underestimated. This can be attributed to the memory buffer chips the POWER7+ architecture uses to connect memory controller and memory DIMMs. CACTI does not model the additional power consumption of these buffer chips. The modeled and measured power consumption for the Xeon system is more accurate. (2) Even though the models do not precisely match the real measurements, they correctly predict the ranking between the two systems.

4.4 Processor Branch Prediction Modeling

In this section, we focus on modeling the performance of one specific processor component, the branch predictor. A branch predictor predicts the outcome of a conditional branch (and the branch target address in the case of unconditional branches) in order to improve the flow in the instruction pipeline. Numerous prediction schemes have been proposed to achieve this goal [131, 97]. In the case of dynamic branch prediction, essentially a branch predictor learns from the execution of branches which can be taken or non-taken. Thus, a predictor can adapt to dynamic changes in branch behavior at application run-time.

Two main ideas that are usually implemented by branch predictors are: (1) a branch outcome can be correlated with other branches' outcomes (global branch correlation), and (2) a branch outcome can be correlated with the last outcomes of the same branch (local branch correlation). Thus, to learn branching outcome patterns, a predictor usually stores the most recent outcomes, globally for all branches, or locally, per branch, or a combination of the two. These predictors are called two-level predictors, because they use a global branch history register (GHR) as first-level and a pattern history table (PHT) indexed by the GHR as second-level. The GHR can contain global or local branching information. The PHT can be indexed using either only the history or a combination of history outcomes with branch address bits (global or per-address indexing). Each combination of GHR and PHT schemes defines a different type of two-level predictor, e.g., global history and global indexing, global history and per-address indexing.

In our work, we do not propose a new design for branch predictors. Our target is to quantify the predictability of branch behaviors in applications in a predictor-independent manner, taking into account at most two hardware parameters, namely, the history size and the pattern table size. By doing so, we significantly abstract from the inner-workings of a particular predictor architecture. Thus, we look for characterization metrics that can quantify how predictable the branching behavior of an application is and how that metric can be converted to an actual branch prediction rate. Our performance prediction methodology is enabled by a hardware- and ISA-agnostic software characterization framework. Therefore, we investigate how to use PISA to characterize predictability and how to derive an analytic model for predicting the branch prediction rate of an application on a specific processor.

We start with an overview of the state-of-the-art branch entropy metric [132] and continue with an in-depth study of how accurately branch entropy models the history size parameter of the hardware predictor on current processors. The branch entropy does not take into account the size of the pattern tables, but only the history size. We also introduce the first method to reverse engineer the history size of the hardware predictor by analyzing the linear correlation between the measurements of branch misprediction rate and the branch entropy estimates. We also provide a first study about the limitations of branch entropy and propose an approach to derive analytic models of the performance of branch predictors.

As an alternative to branch entropy, we propose a more intuitive method for calculating the predictability of application branch behavior. We call this the *max-outcome* prediction method, because the method relies its estimates on the branch outcome that occurs most of the time. We show that this method estimates branch miss rate by keeping the same correlation with real measurements as the branch entropy. However, it increases the accuracy over the branch entropy across different application from 53% to 70%. In addition, the max-outcome method easily models not only the history size of a predictor, but also the pattern table size as hardware parameters. This enables processor and branch predictor designers to perform early design-space exploration for hardware history and pattern table sizes.

For all real measurements in this section we used the Sandy Bridge processor, because the global history size of its branch predictor (32) is publicly available information [63].

4.4.1 Branch Entropy Overview

In the context of application characterization, the entropy is a metric often used to measure the variability of memory addresses accessed during the execution of a program. The concept has also been used in the context of characterizing the application branch behavior [132]. We use this branch entropy concept to study how to analytically model the performance of branch predictors.

Let S be a source of information generating a limited set of symbols. The sequences x_i of length-n generated by S follow a certain discrete probability distribution p over a finite population $\pi = \{x_i\}$, $i \in \{1,2...N\}$, where N is the number of distinct sequences of n symbols generated by S. The entropy $\mathrm{E}(n)$ is then defined as:

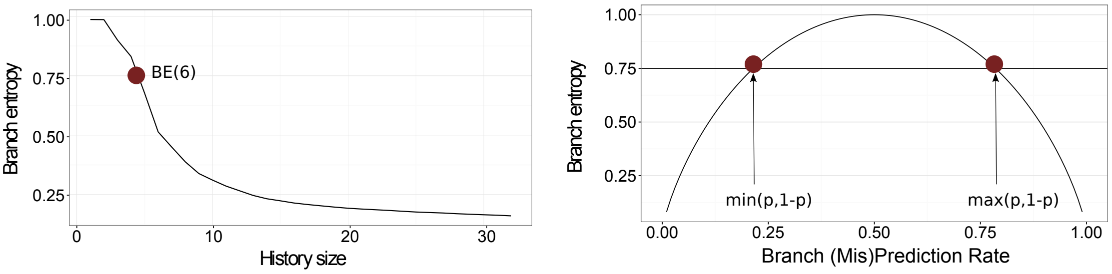
$$\mathrm{E}(n) = -\sum_i p(x_i) \cdot \log_2(p(x_i)). \tag{4.11}$$

The extreme cases of predictability using the entropy concept are: (1) perfect predictability, when the source generates a single sequence which is trivial to predict, $\exists\, x \in P$ such that $p(x) = 1$ and $p(x_i) = 0\ \forall x_i \neq x$, which implies $\mathrm{E}(n) = -\log_2(1) = 0$, and (2) perfect unpredictability, when all sequences of symbols are generated with the same probability, thus, difficult to predict, $\forall x_i\ p(x_i) = \frac{1}{|P|} = \frac{1}{N}$, which implies that $\mathrm{E}(n) = -\sum_1^N \frac{1}{N} \cdot \log_2(\frac{1}{N}) = \log_2(N)$.

The entropy can be leveraged to quantify the predictability of a branch behavior. An application can be seen as a source of branch instructions that generate two symbols 1 and 0, each representing a type of branch outcome, taken or non-taken, respectively. Qualitatively, the entropy of the next branch outcome given access to the n previous outcomes (history size) is the entropy of $(n+1)$-length sequences of S minus the entropy of n-length sequences. Thus, given a history size n, the branch entropy is defined as $\mathrm{BE}(n) = \mathrm{E}(n+1) - \mathrm{E}(n)$, where $\mathrm{BE}(n) \in [0,1]$ and $\mathrm{BE}(n)$ is a monotonically decreasing function with n [132].

To quantify the prediction rate of the outcome of the next branch, given the history of the past n branch outcomes, we need to convert the branch entropy $\mathrm{BE}(n)$ to an actual prediction rate. The source of branch outcomes S characterized by $\mathrm{BE}(n)$ is usually complex, with sequences that may follow some generation rules and others that may be fully random. To analytically derive a prediction rate from the branch entropy of such a source is probably not possible.

Instead, we consider a source S with a simplified symbol generation process, but characterized



(a) Branch entropy curve. (b) Branch entropy to (mis)prediction conversion.

Figure 4.6 – Branch (mis)prediction rate derived from branch entropy.

by the same BE(n) as the one for our original source. We assume this source to generate 1 and 0 symbols following a Bernoulli distribution $D(p)$. The probability to generate 1 is p and the probability to generate 0 is $1-p$. To determine p, we need to solve the equation:

$$\mathrm{BE}(n) = -p \cdot \log_2(p) - (1-p) \cdot \log_2(1-p). \tag{4.12}$$

We observe that if p_1 is a solution of Equation 4.12, then also $1-p_1$ is a solution. Indeed, let's consider that we have a branch entropy for the history size of 6, BE(6) – see Figure 4.6a. Figure 4.6b shows that for BE(6) the equation has two solutions p_1 and $p_2 = 1-p_1$. For either of the two solutions, $\max(p, 1-p) = \max(p_1, 1-p_1) = \max(p_2, 1-p_2) = \max(p_1, p_2)$. In summary, the conversion rule from branch entropy to branch prediction rate is $\max(p, 1-p)$ and for the branch misprediction rate is $1 - \max(p, 1-p) = \min(p, 1-p)$.

Next, we study how well the misprediction rate corresponding to BE(n) correlates with branch prediction rates measured on real processors with a global history size of n. Namely, we compare how the misprediction rate corresponding to BE(32) correlates with real measurements performed on a Sandy Bridge processor. In Subsection 3.3.4 we described what information PISA extracts to calculate the branch entropy. PISA collects the branch outcomes of instructions that change the control flow in a program. We use PISA to calculate the branch entropy and the misprediction rates for a history size of 32 for the SPEC CPU2006 and Graph 500 benchmarks described in Table 4.4.

Given this set of hardware-agnostic entropy-based misprediction estimates, we investigate if we can find a linear model that predicts the real measurement. We create a prediction model for the real measurements by ordinary least-square regression, by minimizing the residual sum of squares, where the residuals are the differences between the measurement and the estimate calculated with branch entropy. The goodness of the fit, the coefficient of determination, is a statistical measure that indicates how well the regression line approximates the measurements. If the coefficient is 1, then the regression line perfectly fits the data. If the coefficient is 0, then there is no correlation between the two sets of values.

Figure 4.7 shows the linear relationship between the real measurements of branch mispredic-

Benchmark	**Domain**	**Input data set**
SPEC CPU2006		
429.MCF	Combinatorial optimization	test
445.GOBMK	Artificial intelligence	test
450.SOPLEX	Linear programming	test
453.POVRAY	Ray-tracing	test
456.HMMER	DNA pattern search	test
458.SJENG	Artificial intelligence	test
462.LIBQUANTUM	Physics (quantum computing)	186
464.H264REF	Video compression	test (10 frames)
482.SPHINX3	Speech recognition	test
Graph 500		
SEQ-LIST	Graph analytics (BFS)	s10-15, e16
SEQ-CSR	Graph analytics (BFS)	s10-15, e16

Table 4.4 – SPEC CPU2006 and Graph 500 applications used in the experiments.

tion performed on a Sandy Bridge processor with the branch-entropy-related misprediction estimates for a history size of 32. The linear correlation factor is 0.78 which indicates a reasonably good linear correlation.

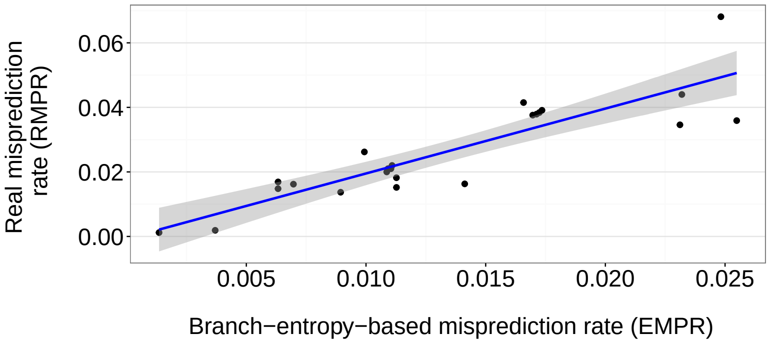


Figure 4.7 – Sandy Bridge real measurements vs. branch-entropy estimates.

4.4.2 Branch Entropy-Based Reverse-Engineering of Hardware Parameters

We obtained a reasonably good correlation between real measurements and branch-entropy misprediction estimates. There are two possible explanations to obtaining a good correlation. One is that the model captures the inherent predictability of the branching behavior of the application. The second is that the model integrates characteristics of the hardware system on which the application is running, e.g. history size (global or local). Thus, to understand to what extent the model is able to cover the two aspects, we performed the following experiment.

For each history size from 1 to 48 we built a separate linear model that estimates the measured branch miss rate from the branch entropy miss rates. Each such linear model has its own coefficient of correlation that we show in Figure 4.8. The global results assume that the source of branch outcomes has a single global history without differentiating between branches. The local results, however, assume that each branch has a local history of the same size. In the global model, the branch entropy is calculated across all branch outcomes, independently of the branch. In the local model, we calculate the branch entropy for each individual branch and the final branch entropy characterizing the source of branches is an average across the local branch entropies weighted with the frequency of each branch.

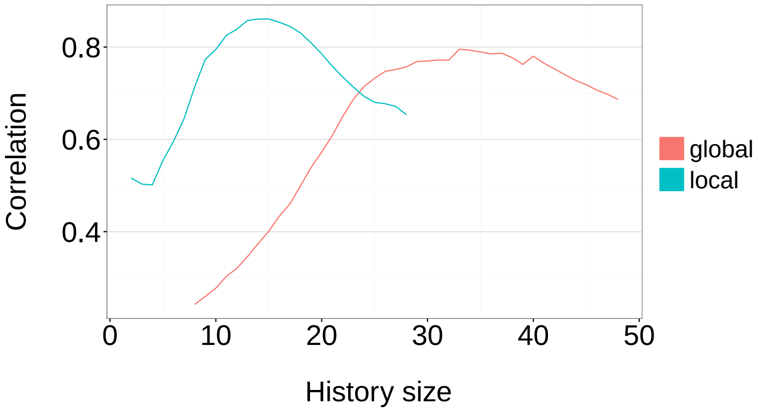


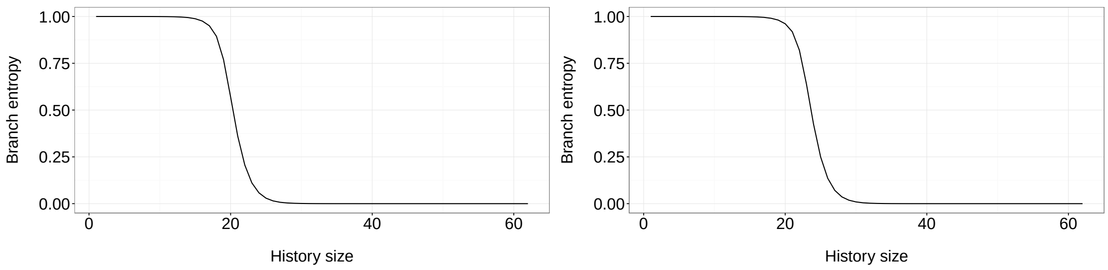
Figure 4.8 – Correlation analysis across history sizes for the Sandy Bridge processor.

The experimental results are encouraging. The global model with one of the highest correlations (0.772) is the one that uses the correct hardware parameter value, the global history size which is 32 for the Sandy Bridge processor [63]. The maximum correlation value was obtained for a history size of 33, with a correlation of 0.795, 2.9% higher than the correlation obtained for the real global history rate. As we artificially introduce error into our model by using higher or lower values for the hardware parameter, the correlation decreases. We conclude that (1) the branch entropy models the hardware parameter of global history size with reasonable accuracy, and (2) the method described above can be used to reverse engineer the history sizes of real branch predictors.

The method was validated not only for Sandy Bridge, but also for POWER processors, for which, however, we do not show results for confidentiality reasons. For the local model we could not validate the local history size, because this information is not public. However, following the results obtained for the global model, if the Sandy Bridge processor would have a local history size, the size could be the one corresponding to the linear model with the highest correlation, namely 14.

4.4.3 Branch Entropy Limitations

Although the branch-entropy estimates for branch miss rate correlate reasonably well with real measurements, we identified a limitation of the branch entropy method, which is a consequence of the limited trace size. To explain this limitation we perform the following



(a) Random-1M.

(b) Random-10M.

Figure 4.9 – Branch entropy for random traces.

experiment. We synthetically generate a trace of branch outcomes using a random generator of 0 and 1 symbols. We calculate for each trace the branch entropy for different history sizes from 1 to 64. Figures 4.9a and 4.9b show the global branch entropy results for two random traces, one with 1 million entries and one with 10 million entries, respectively.

A source with uniform random behavior has a branch entropy of 1. Given that the source of information used in this experiment is random, we would expect the branch entropy graphs to show this random behavior regardless of the history size, namely the branch entropy to constantly be at a value of 1. Instead, the results in the figures show that the branch entropy is able to capture this randomness only up to a certain global history size, roughly 16 and 20, for the 1M-trace and the 10M-trace, respectively. Beyond 16 and 20, the branch entropy quickly drops to a nearly 0-value, which implies that the behavior is highly predictable.

This experiment raises an interesting question: how large the trace size should be so that the samples in the trace are representative of the probability distribution of the source generating the trace. It is not clear how to answer this question, as typically the distribution is dependent on the probability itself, which we do not know.

For example, if $p(x) = i$ for some string x and 0 otherwise, then 1 sample is enough. Otherwise, if the probability distribution is roughly uniform over all strings, then a minimum of $k \cdot 2^n$ samples are necessary, for some value k, to ensure that each of the possible sequences of the distribution occurred at least several times in the trace. This implies that the trace size t should be larger than $k \cdot 2^n$ and, thus, the maximum history size for which the branch entropy is still reliable is $n = \log_2(t) - k$.

To determine the value of k we use the random trace experiment. For the 1M-entry random trace, the history size that still gives a reliable branch entropy of 1 is $16 = \lceil \log_2(10^6) \rceil - 3$. For the 10M-entry random trace, the history size that still gives a reliable branch entropy of 1 is $20 = \lceil \log_2(10^7) \rceil - 3$. Therefore, we decided to use $k = 3$. We note $\mathrm{B\hat{E}}_t(n)$ the most reliable branch entropy of a history size of n, given the trace size t and $\mathrm{B\hat{E}}_t(n)$ corresponds to $n = \log_2(t) - 3$.

We can also theoretically show what may happen if too few samples are available in the trace. Let's say that p is a uniform probability distribution over 2^n sequences of length n and we sample k sequences, $k < 2^n$. The entropy of the source with the distribution p, as it is uniform random, is $E = n$, but as we have only k samples, the entropy of the available samples will be at most $\log_2(k)$, probably even lower due to sequences repeating in a uniform sample.

Let's assume that the branch entropy method is independent of the sample size. Let's also assume that the source is uniform random and that the sample size is $k = 2^m$. In this case $\mathrm{E}(1) = 1$, $\mathrm{E}(2) = 2$ and $\mathrm{E}(m) = m$, where $m = \log_2(k)$. However, $\mathrm{E}(m+1) \leq \log_2(k) = m$ and, thus, $\mathrm{E}(m+1) - \mathrm{E}(m) \leq m - m = 0$. This contradicts the original hypothesis that the source is random and that the branch entropy is independent of the sample size. Thus, ideally one should consider that the most reliable branch entropy value of an application with a limited trace size t is $\mathrm{B\hat{E}}_t(\log_2(t) - 3)$. To the best of our knowledge, the analysis presented in this section is the first to outline such a limitation of the concept of branch entropy.

With the latest outcome of our analysis, to build analytic models of the performance of branch predictors, one could use a methodology as follows. Let's assume we have a set of applications. For each application, we calculate the branch entropy and its corresponding branch misprediction rate for a history size of $\log_2(t) - 3$, where t is the size of the application branch trace. The misprediction rate corresponding to $\mathrm{B\hat{E}}_t(\log_2(t) - 3)$ could be used in an analytic processor performance model to quantify the branch prediction performance.

Alternatively, one could further perform a set of branch misprediction measurements on existing processors for the same applications. Then we build a linear regression model that predicts the real measurement from the hardware-independent misprediction rate. This is a model that could also be used to quantify the misprediction rate in an analytic compute performance model such as the one presented in [77, 76].

In summary: (1) To quantify the inherent branch predictability of an application with a limited branch trace size t, the branch entropy should ideally be calculated for a history size of $\log_2(t) - 3$ to remove inaccuracies due to limited branch trace size; (2) To quantify the branch predictability of an application with a limited branch trace size t, for a branch predictor with a given hardware global history size n, one could use either $\mathrm{BE}(n)$ (as shown in our previous experiments, we obtained a good correlation factor between the real measurements and the entropy estimates corresponding to a global history size of n) or, ideally, $\mathrm{BE}(\log_2(t) - 3)$; (3) To build a model for a real branch predictor, one could use misprediction rates corresponding to a branch entropy of history size of either (ideally) $\log_2(t) - 3$ or n, where n is the global history size of the real branch predictor and t is the branch trace size of the application.

4.4.4 Branch Predictability Max-Outcome Metric Overview

Two important hardware parameters of a branch predictor are the global history and the pattern table sizes. The branch entropy metric models with reasonable accuracy the global history size, however it does not model the capacity of the pattern table. We introduce an analytic method that can model both hardware parameters. Our solution could be used for early design-space exploration of branch predictor architectures. We call this approach the *max-outcome* prediction method, because it estimates the prediction rate per pattern based on the next outcome of the pattern that occurs most of the time.

We consider, as in the case of branch entropy, the branching decisions as a stream of 0 and 1 bits (either global or local per branch decisions). We further consider a fixed history size n, corresponding to the global or local buffer size of a branch predictor. We also assume that for a given length-n branching pattern h, the outcome of the next branch corresponds to a binary stochastic process that outputs 1 with probability p. Then, the best prediction rate we can then obtain is by predicting the most frequent outcome for each pattern, which gives a prediction rate of $\max(p, 1-p)$. The ideal aspect of this prediction rate estimate derives from assuming that the most frequent outcome for each history pattern is known from the start.

To estimate p, for every length-n pattern h, we count the number of times it is followed by a 0 (N_0) and the number of times it is followed by a 1 (N_1). Then $p = \frac{N_1}{N_1+N_0}$ and the ideal prediction rate quantified as $\frac{\max(N_1,N_0)}{(N_1+N_0)}$. The calculation of p is similar to the taken rate metric [47], however, we do not calculate one taken rate per branch instruction, but a taken probability per history pattern of a given length. A corner case we also consider is that when the pattern occurs a single time. In that case no prediction power can be expected of even an ideal predictor. The prediction rate in this case is 0.5. Given the per-history-pattern prediction rates, the global prediction rate is computed as the average across history patterns, weighted by the number of occurrences of each pattern. Table 4.5 shows an example of how the max-outcome method calculates the prediction rates per branching pattern h. The total prediction rate in this case is $\frac{100\cdot 0.7+50\cdot 0.8+20\cdot 0.75+10\cdot 0.7}{100+50+20+10} = 73\%$

Pattern h	N_0	N_1	**Total**	**Prediction rate**
00	70	30	100	70%
01	10	40	50	80%
10	15	5	20	75%
11	3	7	10	70%

Table 4.5 – Example of max-outcome computation (no size limitation on the pattern table).

To account for potentially limited memory available to the predictor (limited pattern table size), we will make the following assumptions. A predictor can only keep track of at most T history patterns and their associated probabilities. For all the other patterns, it will treat them as a single pattern or entry. Thus, for this entry it will estimate a single probability based on the aggregate N_0 and N_1 counts. The ideal aspect derives from keeping track of the T most frequent patterns and knowing them from the start. Table 4.6 shows the same example as the previous table but for a limited pattern table size of $T = 3$ entries. In this case, the total prediction rate will be $\frac{100 \cdot 0.7+50 \cdot 0.8+30 \cdot 0.6}{100+50+30} = 71\%$. As expected by limiting the table size, the prediction rate decreases in comparison with when the table had no limitations on its size.

Pattern h	N_0	N_1	**Total**	**Prediction rate**
00	70	30	100	70%
01	10	40	50	80%
Rest	18	12	30	60%

Table 4.6 – Example of max-outcome computation (limited pattern table size).

4.4.5 Characterization Results

We performed branch misprediction measurements on a Sandy Bridge processor for the SPEC CPU2006 and Graph 500 applications in Table 4.4 (we did not include *462.libquantum* as both the entropy and the max-outcome metrics were significantly overestimating the misprediction rate). We compared these measurements with the hardware-agnostic branch-entropy and max-outcome miss rates. For the latter, in order to make the comparison with the branch entropy fair, we take the case when there are no limitations on the pattern table size. The information about the pattern table sizes is, to the best of our knowledge, not publicly available.

Figure 4.10 shows the correlation results for both the branch entropy and the max-outcome miss rates. The correlation is calculated across applications between the estimates of a given branch behavior predictability metric and the real measurements. As branch-entropy-related metrics, we use the misprediction rate corresponding to the branch entropy calculated for a history size of 32 (equal to the global history size of the Sandy Bridge branch predictor) and the misprediction rate corresponding to the most reliable branch entropy, given the limited branch trace size $\hat{\mathrm{BE}}_t(\log_2(t)-3)$. In the figure we call the latter "branch entropy (ideal)". Moreover, as alternative metrics for quantifying branch predictability, we use the max-outcome metric for history sizes of 32 and $\log_2(t)-3$, where t is the application trace size. We call the latter "max-outcome (ideal)". The $\log_2(t)-3$ value does not have a specific meaning for the max-outcome metric, as it does for the branch entropy metric. We simply present both results to make a fair comparison with the branch entropy values.

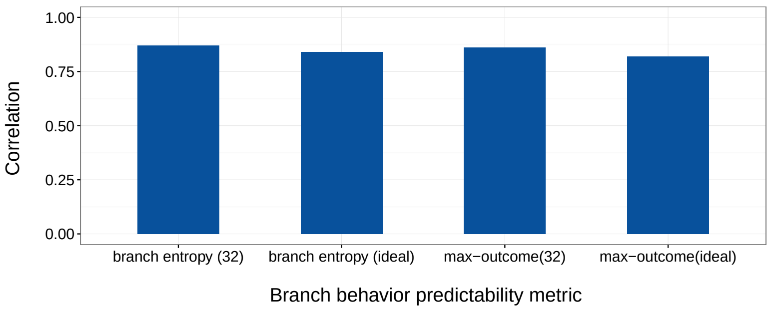


Figure 4.10 – Branch entropy vs. Max-outcome - Correlation across applications.

The results in Figure 4.10 show that the max-outcome metric has a high correlation across applications, similar to branch entropy. Indeed, the max-outcome metric achieves a correlation factor of 0.86 (for a global history size of 32) and 0.82 (for a global history size of $\log_2(t) - 3$). The branch entropy metric exhibits also high correlation across applications, namely, 0.87 (for a global history size of 32) and 0.84 (for a global history size of $\log_2(t) - 3$).

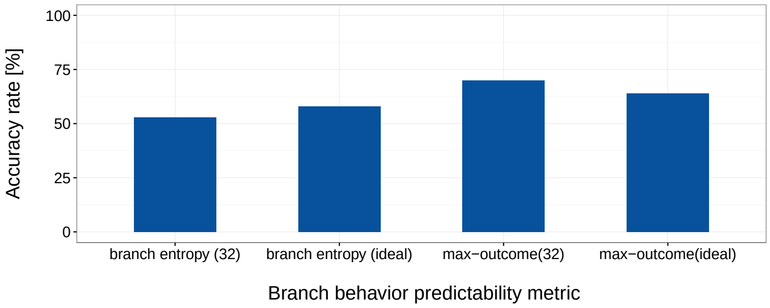


Figure 4.11 – Branch entropy vs. Max-outcome - Accuracy rate across applications.

Furthermore, the model error for an application is calculated as $\frac{|\text{Model}-\text{Measurement}|}{\text{Measurement}}$. Then, we compute the model accuracy as 1 minus the average error across applications. Figure 4.11 shows the accuracy results for both the branch entropy and max-outcome metrics when compared with the real measurements.

The branch entropy metric for a history size of 32 exhibits an average accuracy of 53% (with a maximum accuracy of 92% and a minimum accuracy of 30%). The branch entropy metric for a history size of $\log_2(t) - 3$ exhibits a slightly higher average accuracy of 58% (with a maximum accuracy of 92.7% and a minimum accuracy of 33.8%). The max-outcome metric, however, shows a higher accuracy of 70% (with a maximum accuracy of 98.5% and a minimum accuracy of 11%) for a history size of 32 and 0.64 (with a maximum accuracy of 98% and a minimum accuracy of 33%) for a history size of $\log_2(t) - 3$. The minimum accuracy of 11% seems to be an outlier, as the average accuracy across the remaining applications is 78% with a minimum accuracy of 60% and maximum accuracy of 98.5%.

In conclusion, the max-outcome branch behavior predictability metric is not only able to preserve a good linear correlation across applications as branch entropy, but it is also more accurate on average than branch entropy. The max-outcome metric also models not only the global history size of a branch predictor, but also the table pattern size, thus, allowing processor designers and researchers to perform design-space exploration of the two hardware parameters.

4.5 Related Work

The work presented in this chapter can be compared with previous (1) approaches that model processor performance, and (2) branch predictability metrics and modeling of hardware branch prediction.

4.5.1 Processor Performance Modeling

A wide variety of methods exist in the literature that address the modeling of a processor's performance. Many of these methods include processor simulators [27, 45]. Such methods are usually reasonably accurate in predicting performance, however, they are time-consuming. With execution-based simulators, no software profiles are required, as applications can simply be run on the simulation infrastructure. With trace-based simulators, application traces must be available. For each architectural parameter change, a new simulation is necessary. In our work, we take the inverse approach. We want to analyze the application once in a hardware-agnostic manner and then use that profile with a fast analytic model to explore the performance across different hardware configurations. For our objectives, simulators would not be a feasible solution, as we aim for fast early design-space exploration of a large number of hardware design points. This is relevant especially when we want to model large-scale (exascale) systems.

Processor performance has also been analyzed by using performance data collected via measurements on existing or simulated processor architectures [89, 61, 96]. In our work, we aim at decoupling the software properties from hardware performance modeling and combine them with analytic performance models that describe the interactions between software and hardware.

The closest related work to our modeling objectives is presented by Van den Steen et al. [57]. They also propose to use an analytic processor model with hardware-architecture-agnostic software profiles, but only for x86 profiles. They propose as next steps to perform similar processor performance evaluation using ISA-independent profiles. We tackle precisely this particular topic. We propose the first analysis of the feasibility of analytically analyzing the processor performance based on hardware- and ISA-agnostic software profiles.

4.5.2 Branch Predictability Modeling

Regarding branch predictability metrics, Yokota et al. [132] introduce the concept of branch entropy and investigate the correlation between branch-entropy miss rates and miss rates obtained via simulations. We use the branch entropy concept to perform multiple further analyses. First, we analyze the linear correlation between branch-entropy miss rates obtained from hardware-agnostic branching traces and measurements on real processors and we show that there is a reasonably good linear correlation between the two. By analyzing the correlation, we identify the first method to reverse engineer the history size of a hardware predictor. Finally, we provide a first study about the limitations of branch entropy and propose an approach to derive analytic models of the performance of branch predictors.

As an alternative to branch entropy, we propose the max-outcome metric that takes into account not only the history size of a predictor, but also the pattern table size. The max-outcome metric uses the taken rate metric [47]. The differences are (1) the taken rate is calculated per history pattern and not per static branch instruction, (2) we define a misprediction rate per history pattern, and (3) while Chang et al. [47] use the taken rate to classify branches in application (no branch miss rate modeling), we use the new metric to estimate miss rates. The max-outcome method has an accuracy of up to 17 percentage points better than branch entropy when compared with real miss rates.

Joshi et al. [80] use the fraction of taken branches and the fraction of forward-taken branches in addition to a large set of other software properties to find similarities in benchmarks. De Pestel et al. [107] propose a metric derived from branch entropy which replaces the non-linear branch-entropy-to-miss-rate curve in Figure 4.6b with a linear one. They report higher accuracy results when compared with the estimates based on the original branch entropy concept.

4.6 Conclusions

In this chapter, we presented the first analysis of how hardware- and ISA-agnostic software profiles, such as PISA's, enable processor analytic performance modeling. We showed that by using PISA profiles with analytic compute models that model the processor events independently, we obtain for the execution time metric an average accuracy of 45% across the SPEC CPU2006 and Graph 500 benchmarks when compared with measurements on real processors. We also obtain a good correlation factor across the applications for a given hardware architecture of 0.84. When we load the software profiles into analytic compute models that take into account interactions between the machine events, the results show a higher average accuracy of 34% and a correlation factor of 0.97.

We also showed that PISA enables modeling of power consumption of processors and memories, with good accuracies of 80%+ for POWER7+ processors. While the accuracy for the Xeon processors is lower at only 53% for SPEC CPU2006 and 74% for Graph 500, the models accurately capture the rankings between the two processors from a power consumption perspective. While not as accurate as simulators, the methods presented in this chapter show encouraging results for using fast analytic compute model to perform early design-space exploration of large sets of hardware configurations. This is relevant especially in the context of large-scale (exascale) system modeling.

We also performed a detail analysis of a common characterization metric, the branch entropy. We showed that there is a good linear correlation between branch-entropy miss rates obtained from hardware-agnostic branching traces and measurements on current processors. By analyzing the correlation, we identified the first method to reverse engineer the history size of a hardware predictor. We also provided a first study about the limitations of branch entropy and propose an approach to derive analytic models of the performance of branch predictors. Finally, we introduced the max-outcome branch predictor metric that is able to model not only the history size of a branch predictor, but also the size of the pattern table. When assuming no limitations on the pattern table size, the max-outcome out-performs the estimates of branch miss rates based on branch entropy with up to 17 percentage points.

5 Analytic Modeling of Network Communication Performance

The previous two chapters introduced (i) a hardware-agnostic instrumentation method for extracting software signatures from multi-threaded and multi-process applications, and (ii) three use cases of how the software analysis tool can be used to support and evaluate the performance and power consumption of processors. The compute nodes of a large-scale system will be connected via an interconnection fabric. The fabric performance depends on the interaction of a multitude of parameters, such as the network parameters, the application communication pattern, the routing scheme and the mapping strategy of the MPI processes.

In this chapter, we address the analytic modeling of the performance of network topologies under specific workload scenarios. We propose a performance analysis method that takes into account a network specification and a communication pattern to predict the network effective injection bandwidth. We apply this method to derive analytic models for classes of applications described by three communication patterns representative of HPC workloads: uniform, shift and 2-dimensional nearest-neighbor (2D NNE). As network topologies we model 2-level and 3-level fat-trees, 2-dimensional (2D) HyperX, full-mesh and multi-dimensional tori. We validate the models using high-accuracy network simulators.

5.1 Introduction

System design is the first step towards building a large-scale computer system. To find the optimal design, system architects often need to explore a large space of hardware configurations. Thus, it is important for them to use efficient modeling techniques that analyze the performance under many workload scenarios in a timely manner without hardware prototyping or long-running simulations. In such systems, nodes often exchange data with each other to proceed with their local computation. This makes the interconnection fabric and the resulting inter-node communication bandwidth key components that need to be optimized.

Many research studies have analyzed interconnection fabrics by using network simulators [84, 98]. Other approaches have assessed the network performance using linear programming

(LP) formulations which find the maximum flow that can traverse the network fabric given a specific communication matrix [121]. Network simulations and LP formulations are flexible and usually generic solutions and can be used in principle to analyze any topology and workload scenario. However, both approaches are reasonable in time (minutes or hours per hardware design point) and memory resources only for small network scales [82]. Moreover, the execution time of a max-flow LP formulation may grow exponentially with the number of flows traversing the network. This makes such approaches less feasible for analyzing the performance of large-scale networks, especially within frameworks that aim at performing design-space exploration for large sets of exascale network design points.

Although not as generic as the previously mentioned approaches, in this chapter, we propose analytic communication bandwidth models for classes of communication patterns run on network topologies with even millions of nodes. Our objective is to build a set of analytic compute and communication hardware models that can be used to efficiently evaluate the performance of large-scale systems. Moreover, if a network system architect has to analyze a large hardware design space, by using our proposed analytic communication models, the design space can be rapidly analyzed and reduced to a smaller space with hardware configurations that meet specific requirements. If necessary, the resulting smaller set of design points can be further analyzed in more detail using, e.g., simulators or other computationally-intensive (potentially) more accurate methods.

In Chapter 4 we presented how we can load the PISA characterization results into fast analytic compute models to estimate the performance of a processor. In the following sections we will present a methodology to analytically derive the effective bandwidth of an application. The bandwidth models presented in this section will be used in Chapter 6 to determine the communication time and, thus, the total application execution time. We analytically model the effective bandwidth of a parallel MPI application as a function of four software and hardware parameters that impact the network performance: the application communication pattern, the network topology, the network routing scheme and the mapping of the MPI processes to the hardware compute nodes.

1. Communication Patterns. Many HPC applications exhibit an inter-process communication pattern similar to one of the following: uniform, shift, nearest-neighbor, bit reversal, butterfly, complement, matrix transpose [84]. The long-term objective of this work is to introduce bandwidth models for each of these communication patterns. This would allow fast estimation of the performance of a network topology under the most common HPC patterns. In this thesis we will focus on three of these patterns: uniform, shift and 2-dimensional nearest-neighbor. The bandwidth models for the remaining patterns represent future work.

2. Network Topologies. We focus our analysis on existing network topologies which can also be used in a later stage to validate the estimated execution time of an MPI application with real measurements. We model fat-tree, full-mesh, 2D HyperX and torus network topologies.

Fat-trees are very popular topologies, not only for HPC supercomputing systems, but also for data-centers [18].

3. Routing Schemes. There are two main routing schemes employed in real systems: direct and indirect. With direct routing, all packets traverse the network through the shortest path. With indirect routing, they are sent through indirect paths. More precisely, for each packet, the routing is performed in two steps: (1) an intermediate node different from the destination node of the packet is randomly chosen, and the packet is routed to that intermediate node, and (2) route the packet from the intermediate node to the actual destination of the packet.

Fat-tree and torus systems use shortest-path routing. The full-mesh and 2D HyperX usually use a combination of shortest-path and indirect routing based on the temporal dynamics of the network congestion [19]. In this work, we present analytic models for the shortest-path routing. The analytic modeling of indirect routing and combinations of routing schemes represents future work. Our bandwidth models do not include the switch congestion, because this type of congestion is a transient switch state that depends on the temporal dynamics of the application which we do not include in the current PISA software model.

4. MPI Rank Mapping Strategies. There are research studies that analyze the performance of different mapping strategies for various communication patterns and network topologies [109]. In this work we focus on linear MPI rank mapping, because this is one of the most frequently used strategy in practice. For the 2-dimensional nearest neighbor we actually support a wider set of mapping strategies: we divide the application domain in equal-sized partitions that fully populate sub-domains of the hardware domain. We describe this mapping strategy in Section 5.6.

In Section 5.2 we describe the hardware parameters of the network topologies under study. In Section 5.3 we present the methodology used to estimate the communication bandwidth of a parallel application. We apply this method to derive bandwidth estimators for the uniform, shift and 2-dimensional nearest-neighbor communication patterns in Sections 5.4, 5.5 and 5.6, respectively. We validate the models using high-accuracy network simulations in Section 5.7. In Section 5.8 we discuss the related work of this chapter and we conclude in Section 5.9.

5.2 Network Topologies Overview

In this section, we describe the types of network topologies whose performance we analyze under different communication pattern scenarios.

Full-Mesh Topology. The full-mesh topology is described by two hardware parameters a,p. a represents the total number of switches in the network and p is the number of compute nodes attached to a switch. A full-mesh topology has the characteristic of each switch being connected to each of the other switches in the network. Figure 5.1 shows an example of a full-mesh topology with $a = 6$. Table 5.1 summarizes the full-mesh-related hardware parameters taken into account by our methodology.

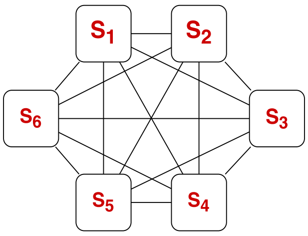


Figure 5.1 – Full-mesh ($a = 6$).

Param.	Meaning
a	no. switches
p	no. nodes per switch
b_0	node-switch bandwidth
b_1	switch-switch bandwidth
l_0	node-switch link latency
l_1	switch-switch link latency

Table 5.1 – Full-mesh parameters.

Fat-Tree Topology. A fat-tree can be described as a multi-stage tree topology where the bandwidth of the connections increases towards the root of the tree. The network topologies modeled in this work belong to the family of extended generalized fat trees (XGFT) [103] which is a general formalization of fat-trees. This family also includes other popular data-center interconnection networks, such as k-ary n-trees and slimmed k-ary n-trees. An XGFT $(h; m_1,..m_h; w_1,..w_h)$ has $h+1$ levels of nodes divided into leaf nodes and inner nodes. There are $\prod_{i=1}^{h} m_i$ leaf nodes occupying level 0 and they serve as end compute nodes. The inner nodes occupy levels 1 to h and serve as switches. Each non-leaf node on level i has m_i child nodes and each non-root node on level j has w_{j+1} parent nodes. w_0 represents the number of links through which an end node is connected to an L1 switch. w_1 represents the number of upward links per L1 switch and so on.

Figure 5.2 shows an example of a 2-level XGFT topology with $w_0 = 2$, $w_1 = 2$, $m_1 = 2$ and $m_2 = 4$. Table 5.2 summarizes the hardware parameters of a 2-level XGFT topology taken into account by our methodology. For a 3-level XGFT, in addition to the parameters in Table 5.2, the network has 4 more parameters: w_2 (the number of upward links per L2 switch), m_3 (the number of downward links per L3 switch), b_2 (the switch-switch bandwidth between levels 2 and 3) and l_2 (the switch-switch link latency between levels 2 and 3).

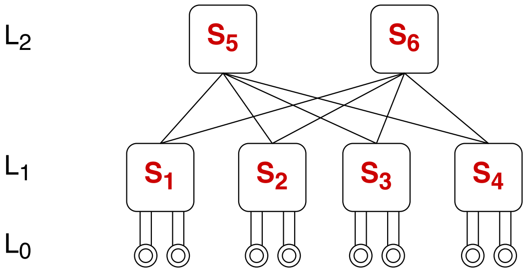


Figure 5.2 – 2-level XGFT (2;2,4;2,2).

Param.	Meaning
w_0	upward links per end node
w_1	upward links per L1-switch
m_1	downward links per L1-switch
m_2	downward links per L2-switch
b_0	node-switch bandwidth
b_1	switch-switch bandwidth
l_0	node-switch link latency
l_1	switch-switch link latency

Table 5.2 – 2-level XGFT parameters.

In Figure 5.3 we show a 3-level XGFT with $w_0 = 2$, $w_1 = 2$, $w_2 = 4$, $m_1 = 2$, $m_2 = 2$ and $m_3 = 2$. This topology is functionally equivalent to the traditional fat-tree in Figure 5.4. At levels 0 and

1, the traditional fat-tree and the XGFT are identical in structure. Moving up into the tree, in the XGFT we have single links between connected switches, but a switch can have multiple next-level switch neighbors. Conversely, in the traditional fat-tree we always have a single next-level neighbor, but multiple links to it. Indeed, in the traditional fat-tree, every L1 switch has w_1 upward links, all going to its single L2 neighbor. Also, every L2 switch has $w_1 \cdot w_2$ upward links [103], all going to its single L3 neighbor. In our bandwidth models, given its simplicity and functional equivalence to XGFTs, we will use the traditional fat-tree representation.

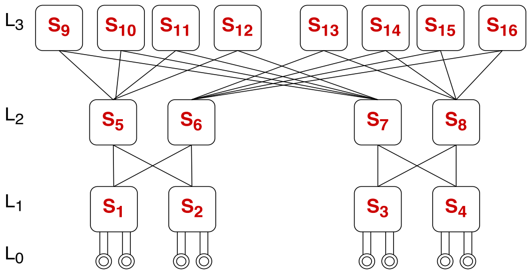


Figure 5.3 – 3-level XGFT (3;2,2,2;2,2,4).

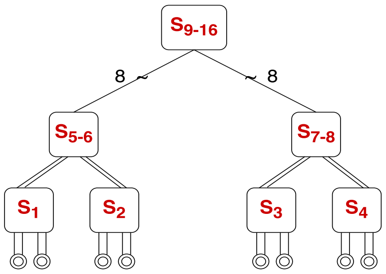


Figure 5.4 – 3-level traditional fat-tree.

2-Dimensional HyperX Topology. The N-dimensional HyperX interconnect fabric is a topology resulting from the Cartesian product of n fully-connected graphs of switches [16, 37]. The 2D HyperX is a 2-level hierarchical topology. It is described by three hardware parameters p,d_1,d_2. p is the number of nodes connected to a switch. d_1 and d_2 represent the numbers of switches in the X and Y dimensions, respectively. The switches in the X dimension are connected via a full-mesh topology. The same applies for the switches in the Y dimension. Figure 5.5 shows an example of a 2D HyperX topology with $d_1 = 4$ and $d_2 = 3$. Table 5.3 summarizes the hardware parameters of this topology taken into account by our methodology.

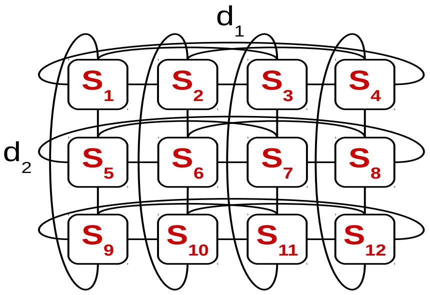


Figure 5.5 – 2D HyperX ($d_1 = 4$, $d_2 = 3$).

Param.	Meaning
p	nodes per switch
d_1	switches in dimension 1 (X)
d_2	switches in dimension 2 (Y)
b_0	node-switch bandwidth
b_1	switch-switch bandwidth (X)
b_2	switch-switch bandwidth (Y)
l_0	node-switch link latency
l_1	switch-switch link latency (X)
l_2	switch-switch link latency (Y)

Table 5.3 – 2D HyperX parameters.

Torus Topology. The torus topology can be seen as a mesh interconnect with nodes arranged in arrays of 2 or more dimensions. An N-dimensional torus has n dimensions, each node has $2 \cdot n$ neighbors and communication can take place in $2 \cdot n$ directions. Such network topologies have been widely used in BlueGene supercomputers [51, 14]. Figure 5.6 shows an example of a 2-dimensional (2D) torus with $d_1 = 4$ and $d_2 = 3$. Table 5.4 summarizes the hardware parameters of a 2D torus taken into account by our methodology.

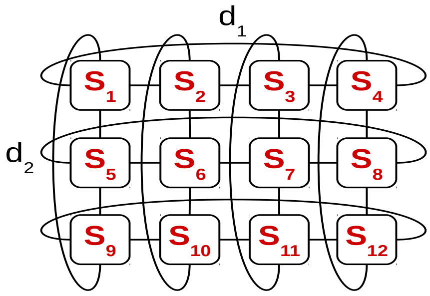


Figure 5.6 – 2D torus ($d_1 = 4$, $d_2 = 3$).

Param.	Meaning
p	nodes per switch
d_1	switches in dimension 1 (X)
d_2	switches in dimension 2 (Y)
b_0	node-switch bandwidth
b_1	switch-switch bandwidth (X)
b_2	switch-switch bandwidth (Y)
l_0	node-switch link latency
l_1	switch-switch link latency (X)
l_2	switch-switch link latency (Y)

Table 5.4 – 2D torus parameters.

5.3 Communication Bandwidth Modeling Methodology

Our end goal is to estimate the communication time of an MPI application in a fast manner. To achieve this goal we need the amount of data exchanged between any two communicating pairs of processes and the bandwidth at which the messages are exchanged. Analytically calculating the node bandwidth for an arbitrary communication pattern is a complex problem due to the large number of (temporal) system inter-dependencies determining the bandwidth. HPC applications, however, usually have a regular communication pattern, which we leverage to derive fast analytic bandwidth models.

Given an MPI application, we first characterize with PISA its communication properties in a hardware-independent manner, by extracting its communication matrix—the amount of data exchanged between any pair of processes during the application runtime. Based on this matrix, we categorize it (when possible) as one of the regular patterns representative of HPC applications, e.g., uniform, shift, 2-dimensional nearest-neighbor. Then, we estimate the application bandwidth as the bandwidth of its equivalent regular communication pattern, making two assumptions: (1) regardless of how many cores an end node has, there is one single-thread MPI process mapped per node, and (2) the MPI processes of the application fully populate the system (each node runs one MPI process).

Communication-wise an MPI application is a set of communicating processes. We call the data transfer occurring between two given processes a *flow*. For the remainder of this chapter, whenever we refer to a flow between two compute nodes, we refer to the flow between the two processes assigned to those nodes. To determine the effective bandwidth of the application we first determine the bandwidth available to each flow. So, with our method, we provide bandwidth estimates at two granularities, flow and application. From the latter, one can also derive a global estimator for the effective bandwidth per node, by dividing the application bandwidth by the total number of nodes.

We make two assumptions when deriving the per-flow bandwidth models:

1. Fairness of link bandwidth allocation. If multiple flows share the same link, then each flow receives the same fraction of the available bandwidth. For example, in Figure 5.7 the link L_2 is shared by two flows. Each one receives half of the available bandwidth.
2. Maximum bottleneck across traversed links. If a flow traverses multiple links, then the effective bandwidth seen by the flow is the effective bandwidth achieved on the link with the highest contention. In Figure 5.7 the green flow between the hosts H_2 and H_4 traverses three links, L_1, L_2 and L_3. Assuming that all the links have the same physical bandwidth, the effective bandwidth of the green flow is given by the bandwidth obtained on the link with the highest contention L_2.

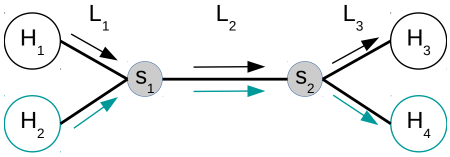


Figure 5.7 – Link contention example illustrating the bandwidth modeling assumptions.

We define the effective application bandwidth as the total volume of data transfered through the network divided by the time necessary to put the data on wire (the wire time) as shown in Equation 5.1).

$$B^{\text{eff}}_{\text{app}} = \frac{\text{Volume}_{\text{app}}}{\text{Time}^{\text{wire}}_{\text{Volume}_{\text{app}}}} \tag{5.1}$$

Once we have derived the bandwidth of each flow we determine the effective application bandwidth as follows. The total data volume of the application is the aggregate volume exchanged across all flows. The time to put this volume of data on wire for the entire application is the longest such time across the individual flows. For a given flow j the wire time is calculated as the volume of data of that flow Volume_j divided by its effective bandwidth B^{eff}_j. Thus, the effective bandwidth of the application is calculated as in Equation 5.2.

$$B^{\text{eff}}_{\text{app}} = \frac{\sum_{\text{i}} \text{Volume}_{\text{i}}}{\max_{\text{j}} \frac{\text{Volume}_{\text{j}}}{B^{\text{eff}}_{\text{j}}}} \tag{5.2}$$

The (average) effective bandwidth per process is calculated as:

$$B^{\text{eff}}_{\text{node}} = \frac{B^{\text{eff}}_{\text{app}}}{\text{n}} \tag{5.3}$$

where n is the number of nodes (which equals the number of processes) in the system.

As our bandwidth methodology computes the bandwidth of every single flow, it could be used to provide estimators at multiple granularities: (i) an individual bandwidth for every flow, (ii) an individual bandwidth for every node, or (iii) one bandwidth value for the entire application. However, not all granularities are compatible with the current full-system performance prediction methodology.

Indeed, as shown in Chapter 6, our full-system prediction method decouples the software properties from hardware performance modeling and combines the hardware and software properties of a system through fast analytic models to allow the evaluation of large hardware design spaces. The software profiler (PISA) extracts compute and communication signatures from the MPI applications, per MPI process. The extrapolation tool (ExtraX) takes the PISA software profiles at small scales and extrapolates them to large scales. These two components of the method (PISA and ExtraX) are decoupled from the hardware modeling.

Before the actual extrapolation, ExtraX first clusters the MPI ranks in classes of processes with similar compute (instruction mix, instruction-level parallelism, data memory reuse pattern) and communication (total amount of data exchanged) properties. Then, by employing machine-learning techniques, ExtraX learns from a set of software profiles how the different compute and communication properties scale to larger problem sizes. ExtraX uses the extrapolation model to predict the profiles of the identified classes at target scale.

In order to use the highest bandwidth granularity (per-flow) with the extrapolated profile, our full-system methodology would need to extrapolate the entire communication matrix (not only the total amount of data exchanged) which is currently not supported. To use the medium bandwidth granularity (per process or node), the methodology would need to preserve in the extrapolation the mapping of processes to the application domain. This is currently supported by ExtraX, however, our bandwidth models would become exceedingly detailed. Therefore, for the remainder of the thesis, we will work at the third granularity level and use the single estimator $B_{\text{node}}^{\text{eff}}$ (derived from the effective application bandwidth) as the bandwidth at which the nodes send data through the network.

5.4 Bandwidth Models: Uniform Communication Pattern

In a parallel application with uniform communication pattern, each MPI process sends data to each of the other nodes including to itself with the same probability. Each MPI process generates n data flows, where n is the total number of MPI processes of the application. On average each of these flows transports the same volume of data V.

5.4.1 Full-Mesh Topology

The total number of compute nodes in a full-mesh topology is $a \cdot p$ and the number of flows per compute node in the case of a uniform communication pattern is $a \cdot p$. Thus, the total number of flows generated by the application through the network is $(a \cdot p)^2$.

We define a flow type as a set of flows sharing the same path characteristics (they traverse the same kinds of links in the same order). Each flow within a flow type will have the same effective bandwidth. However, two flow types can potentially have two different effective bandwidths. For the uniform pattern run on a full-mesh topology we identify two flow types. The first type includes flows between communicating nodes that are connected to the same switch, whereas the second type includes flows between nodes connected to different switches.

Flows of the first type send messages through two links: source-switch and switch-destination. The effective bandwidth of this first flow type is calculated as the minimum between the two effective bandwidths achieved on each of the links:

$$B_1 = \min(B_{\text{source-sw}}, B_{\text{sw-destination}}). \tag{5.4}$$

Flows of the second flow type sends the messages through three links: source-switch, switch-switch and switch-destination. In this case, the effective bandwidth is calculated as:

$$B_2 = \min(B_{\text{source-sw}}, B_{\text{sw}-\text{sw}}, B_{\text{sw-destination}}). \tag{5.5}$$

By applying Equation 5.2, we calculate the application bandwidth as in Equation 5.6.

$$B_{\text{app}}^{\text{eff}} = \frac{(a \cdot p)^2 \cdot V}{\max(\frac{V}{B_1}, \frac{V}{B_2})} = (a \cdot p)^2 \cdot \min(B_1, B_2) \tag{5.6}$$

To calculate B_1 and B_2 we analyze the contention on the three types of (directional) links: source-switch, switch-switch and switch-destination. The source-switch link is shared by all the flows that carry messages sent by the source node to all the other nodes in the network. Therefore, $B_{\text{source-sw}}$ is $\frac{b_0}{a \cdot p}$. The switch-destination link is shared by all the flows that carry messages to the destination node from all the other nodes in the network. Thus, $B_{\text{sw-destination}}$ is $\frac{b_0}{a \cdot p}$. Consequently, $B_1 = \min(\frac{b_0}{a \cdot p}, \frac{b_0}{a \cdot p}) = \frac{b_0}{a \cdot p}$.

Moreover, for any given link connecting two switches, say connecting switch S_1 to S_2, the link is traversed only by the flows generated by the p nodes connected to S_1 towards the p nodes of S_2. Thus, the switch-to-switch link is shared by p^2 flows. Therefore, $B_2 = \min(\frac{b_0}{a \cdot p}, \frac{b_1}{p^2})$. Equation 5.7 shows the model for the effective application bandwidth.

$$B_{\text{app}}^{\text{eff}} = (a \cdot p)^2 \cdot \min(\frac{b_0}{a \cdot p}, \min(\frac{b_0}{a \cdot p}, \frac{b_1}{p^2})) = a \cdot p \cdot \min(b_0, \frac{b_1 \cdot a}{p}) \tag{5.7}$$

To determine the effective bandwidth per node for the uniform communication pattern, in a full-mesh topology with shortest-path routing and linear mapping, we use the model shown in Equation 5.8, where the parameters are explained in Table 5.1.

$$B_{\text{node}}^{\text{eff}} = \frac{B_{\text{app}}^{\text{eff}}}{a \cdot p} = \min(b_0, \frac{b_1 \cdot a}{p}) \tag{5.8}$$

5.4.2 2-Level Fat-Tree Topology

The total number of compute nodes in the 2-level fat-tree topology is $m_1 \cdot m_2$, and, in the case of the uniform communication pattern, the number of flows per compute node is $m_1 \cdot m_2$. Thus, the total number of flows generated by the application through the network is $(m_1 \cdot m_2)^2$.

For the uniform pattern run on a 2-level fat-tree topology we identify two flow types traversing the network. The first type includes flows between communicating nodes that are connected to the same L1 switch (switch$_{\text{L1}}$), whereas the second type includes flows between nodes connected to different L1 switches.

The flows of the first type send messages through two links: source-switch$_{\text{L1}}$ and switch$_{\text{L1}}$-destination. The effective bandwidth of this first flow type is calculated as the minimum between the two effective bandwidths achieved on each of the links:

$$B_1 = \min(B_{\text{source}-\text{sw}_{\text{L1}}}, B_{\text{sw}_{\text{L1}}-\text{destination}}). \tag{5.9}$$

The flows of the second flow type send messages through four links: source-switch$_{\text{L1}}$, switch$_{\text{L1}}$-switch$_{\text{L2}}$, switch$_{\text{L2}}$-switch$_{\text{L1}}$ and switch$_{\text{L1}}$-destination. In this case, the effective bandwidth is calculated as:

$$B_2 = \min(B_{\text{source}-\text{sw}_{\text{L1}}}, B_{\text{sw}_{\text{L1}}-\text{sw}_{\text{L2}}}, B_{\text{sw}_{\text{L2}}-\text{sw}_{\text{L1}}}, B_{\text{sw}_{\text{L1}}-\text{destination}}). \tag{5.10}$$

By applying Equation 5.2, we calculate the application bandwidth as in Equation 5.11.

$$B_{\text{app}}^{\text{eff}} = \frac{(m_1 \cdot m_2)^2 \cdot V}{\max(\frac{V}{B_1}, \frac{V}{B_2})} = (m_1 \cdot m_2)^2 \cdot \min(B_1, B_2) \tag{5.11}$$

To calculate B_1 and B_2 we analyze the contention on the four types of (directional) links. The w_0 source-switch$_{\text{L1}}$ links are shared by all the flows that carry messages sent by a source node to all the other nodes in the network. Therefore, $B_{\text{source}-\text{sw}_{\text{L1}}}$ is $\frac{b_0 \cdot w_0}{m_1 \cdot m_2}$. The w_0 switch$_{\text{L1}}$-destination links are shared by all the flows that carry messages to a destination node from all the other nodes in the network. Therefore, $B_{\text{sw}_{\text{L1}}-\text{destination}}$ is $\frac{b_0 \cdot w_0}{m_1 \cdot m_2}$. Consequently, $B_1 = \min(\frac{b_0 \cdot w_0}{m_1 \cdot m_2}, \frac{b_0 \cdot w_0}{m_1 \cdot m_2}) = \frac{b_0 \cdot w_0}{m_1 \cdot m_2}$.

Moreover, the m_1 nodes connected to a switch$_{\text{L1}}$ all share the w_1 up-links from switch$_{\text{L1}}$ to switch$_{\text{L2}}$. All these w_1 links carry flows from the m_1 sources attached to a switch$_{\text{L1}}$ to all the other $m_1 \cdot m_2 - m_1$ nodes in the network. The total number of flows traversing the w_1 links is $m_1 \cdot (m_1 \cdot m_2 - m_1) = m_1^2 \cdot (m_2 - 1)$. Therefore, $B_{\text{sw}_{\text{L1}}-\text{sw}_{\text{L2}}} = \frac{b_1 \cdot w_1}{m_1^2 \cdot (m_2 - 1)}$.

To determine $B_{\text{sw}_{\text{L2}}-\text{sw}_{\text{L1}}}$ we perform a similar calculation. Given a destination node d attached to switch$_{\text{L1}}$, there are $m_1 \cdot m_2 - m_1$ flows with destination d that will traverse the switch$_{\text{L2}}$-switch$_{\text{L1}}$ down-link. As there are m_1 nodes attached to switch$_{\text{L1}}$, the total number of flows traversing the switch$_{\text{L2}}$-switch$_{\text{L1}}$ down-link is $m_1 \cdot (m_1 \cdot m_2 - m_1) = m_1^2 \cdot (m_2 - 1)$. Consequently, $B_{\text{sw}_{\text{L2}}-\text{sw}_{\text{L1}}} = \frac{b_1 \cdot w_1}{m_1^2 \cdot (m_2 - 1)} = B_{\text{sw}_{\text{L1}}-\text{sw}_{\text{L2}}}$ and $B_2 = \min(\frac{b_0 \cdot w_0}{m_1 \cdot m_2}, \frac{b_1 \cdot w_1}{m_1^2 \cdot (m_2 - 1)})$.

Given B_1 and B_2, the model for the effective application bandwidth in Equation 5.11 is equivalent to the model shown in Equation 5.12.

$$B_{\text{app}}^{\text{eff}} = (m_1 \cdot m_2)^2 \cdot \min(B_1, B_2) = m_1 \cdot m_2 \cdot \min(b_0 \cdot w_0, \frac{b_1 \cdot w_1}{m_1 \cdot (1 - \frac{1}{m_2})}) \tag{5.12}$$

To determine the effective bandwidth per node for the uniform communication pattern, in a 2-level fat-tree topology, with shortest-path routing and linear mapping, we use the model shown in Equation 5.13, where the parameters are explained in Table 5.2.

$$B_{\text{node}}^{\text{eff}} = \frac{B_{\text{app}}^{\text{eff}}}{m_1 \cdot m_2} = \min(b_0 \cdot w_0, \frac{b_1 \cdot w_1}{m_1 \cdot (1 - \frac{1}{m_2})}) \tag{5.13}$$

5.4.3 3-Level Fat-Tree Topology

The total number of compute nodes in the 3-level fat-tree topology is $m_1 \cdot m_2 \cdot m_3$, and, in the case of the uniform communication pattern, the number of flows per compute node is $m_1 \cdot m_2 \cdot m_3$. Thus, the total number of flows generated by the application through the network is $(m_1 \cdot m_2 \cdot m_3)^2$.

For the uniform pattern run on a 3-level fat-tree topology we identify three flow types traversing the network. The first type includes flows between communicating nodes that are connected to the same L1 switch (switch$_{\text{L1}}$). These flows do not cross the L1 network level. The second type includes flows between nodes connected to different L1 switches and that do not cross the L2 level. Finally the third type includes the remaining flows, those that reach the L3 level.

The flows of the first type send messages through two links: source-switch$_{\text{L1}}$ and switch$_{\text{L1}}$-destination. The effective bandwidth of this first flow type is calculated as the minimum between the two effective bandwidths achieved on each of the links:

$$B_1 = \min(B_{\text{source}-\text{sw}_{\text{L1}}}, B_{\text{sw}_{\text{L1}}-\text{destination}}). \tag{5.14}$$

The flows of the second flow type send the messages through four links: source-switch$_{\text{L1}}$, switch$_{\text{L1}}$-switch$_{\text{L2}}$, switch$_{\text{L2}}$-switch$_{\text{L1}}$ and switch$_{\text{L1}}$-destination. In this case, the effective bandwidth is calculated as:

$$B_2 = \min(B_{\text{source}-\text{sw}_{\text{L1}}}, B_{\text{sw}_{\text{L1}}-\text{sw}_{\text{L2}}}, B_{\text{sw}_{\text{L2}}-\text{sw}_{\text{L1}}}, B_{\text{sw}_{\text{L1}}-\text{destination}}). \tag{5.15}$$

The flows of the third flow type send the messages through six links: source-switch$_{\text{L1}}$, switch$_{\text{L1}}$-switch$_{\text{L2}}$, switch$_{\text{L2}}$-switch$_{\text{L3}}$, switch$_{\text{L3}}$-switch$_{\text{L2}}$, switch$_{\text{L2}}$-switch$_{\text{L1}}$ and switch$_{\text{L1}}$-destination. In this case, the effective bandwidth is calculated as:

$$B_3 = \min(B_{\text{source}-\text{sw}_{\text{L1}}}, B_{\text{sw}_{\text{L1}}-\text{sw}_{\text{L2}}}, B_{\text{sw}_{\text{L2}}-\text{sw}_{\text{L3}}}, B_{\text{sw}_{\text{L3}}-\text{sw}_{\text{L2}}}, B_{\text{sw}_{\text{L2}}-\text{sw}_{\text{L1}}}, B_{\text{sw}_{\text{L1}}-\text{destination}}). \tag{5.16}$$

By applying Equation 5.2, we calculate the application bandwidth as in Equation 5.17.

$$B_{\text{app}}^{\text{eff}} = \frac{(m_1 \cdot m_2 \cdot m_3)^2 \cdot V}{\max(\frac{V}{B_1}, \frac{V}{B_2}, \frac{V}{B_3})} = (m_1 \cdot m_2 \cdot m_3)^2 \cdot \min(B_1, B_2, B_3) \tag{5.17}$$

To calculate B_1, B_2 and B_3 we analyze the contention on the six types of (directional) links. The w_0 source-switch$_{\text{L1}}$ links are shared by all the flows that carry messages sent by a source node to all the other nodes in the network. Therefore, $B_{\text{source}-\text{sw}_{\text{L1}}}$ is $\frac{b_0 \cdot w_0}{m_1 \cdot m_2 \cdot m_3}$. The w_0 switch$_{\text{L1}}$-destination links are shared by all the flows that carry messages to a destination node from all the other nodes in the network. Therefore, $B_{\text{sw}_{\text{L1}}-\text{destination}}$ is $\frac{b_0 \cdot w_0}{m_1 \cdot m_2 \cdot m_3}$. Consequently, $B_1 = \min(\frac{b_0 \cdot w_0}{m_1 \cdot m_2 \cdot m_3}, \frac{b_0 \cdot w_0}{m_1 \cdot m_2 \cdot m_3}) = \frac{b_0 \cdot w_0}{m_1 \cdot m_2 \cdot m_3}$.

Moreover, the m_1 nodes connected to a switch$_{\text{L1}}$ all share the w_1 up-links from switch$_{\text{L1}}$ to switch$_{\text{L2}}$. All these w_1 links carry flows from the m_1 sources attached to a switch$_{\text{L1}}$ to all the other $m_1 \cdot m_2 \cdot m_3 - m_1$ nodes in the network. The total number of flows traversing the w_1 links is $m_1 \cdot (m_1 \cdot m_2 \cdot m_3 - m_1) = m_1^2 \cdot (m_2 \cdot m_3 - 1)$. Therefore, $B_{\text{sw}_{\text{L1}}-\text{sw}_{\text{L2}}} = \frac{b_1 \cdot w_1}{m_1^2 \cdot (m_2 \cdot m_3 - 1)}$.

To determine $B_{\text{sw}_{\text{L2}}-\text{sw}_{\text{L1}}}$ we perform a similar calculation. Given a destination node d attached to switch$_{\text{L1}}$, there are $m_1 \cdot m_2 \cdot m_3 - m_1$ flows with destination d that will traverse the switch$_{\text{L2}}$-switch$_{\text{L1}}$ down-link. As there are m_1 nodes attached to switch$_{\text{L1}}$, the total number of flows traversing the switch$_{\text{L2}}$-switch$_{\text{L1}}$ down-link is $m_1 \cdot (m_1 \cdot m_2 \cdot m_3 - m_1) = m_1^2 \cdot (m_2 \cdot m_3 - 1)$. Consequently, $B_{\text{sw}_{\text{L2}}-\text{sw}_{\text{L1}}} = \frac{b_1 \cdot w_1}{m_1^2 \cdot (m_2 \cdot m_3 - 1)} = B_{\text{sw}_{\text{L1}}-\text{sw}_{\text{L2}}}$ and $B_2 = \min(\frac{b_0 \cdot w_0}{m_1 \cdot m_2 \cdot m_3}, \frac{b_1 \cdot w_1}{m_1^2 \cdot (m_2 \cdot m_3 - 1)})$.

We further quantify $B_{\text{sw}_{\text{L2}}-\text{sw}_{\text{L3}}}$ and $B_{\text{sw}_{\text{L3}}-\text{sw}_{\text{L2}}}$.

The $w_1 \cdot w_2$ switch$_{\text{L2}}$-switch$_{\text{L3}}$ up-links are shared by $m_1 \cdot m_2 \cdot (m_1 \cdot m_2 \cdot (m_3 - 1))$ flows. Indeed, each of the $m_1 \cdot m_2$ nodes in a switch$_{L2}$ sub-tree send flows to all the nodes in the other switch$_{\text{L2}}$-rooted sub-trees of the network, $m_1 \cdot m_2 \cdot (m_3 - 1)$ nodes. This implies that $B_{\text{sw}_{\text{L2}}-\text{sw}_{\text{L3}}} = \frac{b_2 \cdot w_1 \cdot w_2}{m_1^2 \cdot m_2^2 \cdot (m_3 - 1)}$.

Furthermore, given a switch$_{\text{L2}}$ sub-tree, each of the nodes in the sub-tree (in number of $m_1 \cdot m_2$) will receive flows on the switch$_{\text{L3}}$-switch$_{\text{L2}}$ w_1 down-links from all the other nodes in the network (in number of $m_1 \cdot m_2 \cdot (m_3 - 1)$). Thus, $B_{\text{sw}_{\text{L3}}-\text{sw}_{\text{L2}}} = \frac{b_2 \cdot w_1 \cdot w_2}{m_1^2 \cdot m_2^2 \cdot (m_3 - 1)} = B_{\text{sw}_{\text{L2}}-\text{sw}_{\text{L3}}}$.

Consequently, $B_3 = \min(\frac{b_0 \cdot w_0}{m_1 \cdot m_2 \cdot m_3}, \frac{b_1 \cdot w_1}{m_1^2 \cdot (m_2 \cdot m_3 - 1)}, \frac{b_2 \cdot w_1 \cdot w_2}{m_1^2 \cdot m_2^2 \cdot (m_3 - 1)})$.

Given B_1, B_2 and B_3, the model for the effective application bandwidth shown in Equation 5.17 is equivalent to the model in Equation 5.18.

$$B_{\text{app}}^{\text{eff}} = (m_1 \cdot m_2 \cdot m_3)^2 \cdot \min(B_1, B_2, B_3) = m_1 \cdot m_2 \cdot m_3 \cdot \min(b_0 \cdot w_0, \frac{b_1 \cdot w_1}{m_1 \cdot (1 - \frac{1}{m_2 \cdot m_3})}, \frac{b_2 \cdot w_1 \cdot w_2}{m_1 \cdot m_2 \cdot (1 - \frac{1}{m_3})}) \tag{5.18}$$

To determine the effective bandwidth per node under the uniform communication pattern, in a 3-level fat-tree topology with shortest-path routing and linear mapping, we use the model shown in Equation 5.19, where the parameters are explained in Table 5.2.

$$B_{\text{node}}^{\text{eff}} = \frac{B_{\text{app}}^{\text{eff}}}{m_1 \cdot m_2 \cdot m_3} = \min(b_0 \cdot w_0, \frac{b_1 \cdot w_1}{m_1 \cdot (1 - \frac{1}{m_2 \cdot m_3})}, \frac{b_2 \cdot w_1 \cdot w_2}{m_1 \cdot m_2 \cdot (1 - \frac{1}{m_3})}) \tag{5.19}$$

5.4.4 1-Dimensional Torus Topology

The 1-dimensional (1D) torus is a ring topology. Typically the torus topology implements the dimension-order routing (first route in the X dimension, second route in the Y dimension, and so on) with shortest-path within each dimension. As the torus in this section has one dimension only, the routing scheme is essentially shortest-path in a ring.

If the size of the ring d_1 is an even number, then for every switch S there are two paths of equal length $\frac{d_1}{2}$ to access the switch at $\frac{d_1}{2}$ hops away from switch S. In this case, we will assume that the flows originating in S with destination nodes attached to the switch at $\frac{d_1}{2}$ hops away from S are equally distributed between the two paths.

The total number of compute nodes in the 1D torus is $p \cdot d_1$. The number of flows per compute node is thus $p \cdot d_1$ and the total number of flows generated by the application through the network is $(p \cdot d_1)^2$. For the uniform pattern run on a 1D torus we identify two flow types that traverse the network. The first type includes flows between communicating nodes that are connected to the same switch, whereas the second type includes flows between nodes connected to different switches.

The flows of the first type send messages through two links: source-switch and switch-destination. The effective bandwidth of the first flow type is calculated as the minimum between the two effective bandwidths achieved on each of the links:

$$B_1 = \min(B_{\text{source-sw}}, B_{\text{sw-destination}}). \tag{5.20}$$

The flows of the second type send the messages through at least three links: source-switch, switch-switch and switch-destination. Depending on the locations of the source and destination nodes, the flows traverse one or more consecutive switch-switch links. However, due to the topology and communication pattern symmetries, all the switch-switch links will have the same flow contention. We use Figure 5.8 to show how we calculate the contention on a particular switch-switch link. The link switch$_4$-switch$_0$ (the wrap-around link) has been omitted for simplicity of drawing.

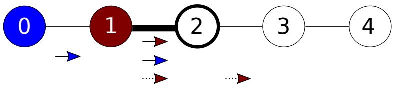


Figure 5.8 – 1-dimensional torus switch-switch link contention.

Let's quantify the number of flows traversing the directional link from switch$_1$ to switch$_2$.

The link is traversed by p^2 flows from the nodes of switch$_1$ to the nodes of switch$_2$ and by p^2 flows from the nodes of switch$_0$ to the nodes of switch$_2$. In addition, due to the routing scheme, the p^2 flows of switch$_1$ to the nodes of switch$_3$ will also traverse the link from switch$_1$ to switch$_2$. Therefore, in total, the link from switch$_1$ to switch$_2$ will be shared by $3 \cdot p^2$ flows. The same calculation can be performed for any link from switch$_k$ to switch$_{(k+1) \bmod d_1}$, for any $k \in [0, d_1 - 1]$ and the result will be the same.

Regardless of how many switch-switch links the flows of the second type traverse, the effective bandwidth of the second type of flows is calculated as:

$$B_2 = \min(B_{\text{source-sw}}, B_{\text{sw}-\text{sw}}, B_{\text{sw-destination}}). \tag{5.21}$$

By applying Equation 5.2, we calculate the application bandwidth as in Equation 5.22.

$$B_{\text{app}}^{\text{eff}} = \frac{(p \cdot d_1)^2 \cdot V}{\max(\frac{V}{B_1}, \frac{V}{B_2})} = (p \cdot d_1)^2 \cdot \min(B_1, B_2) \tag{5.22}$$

To calculate B_1 and B_2 we analyze the contention on the three types of (directional) links. The source-switch link is shared by all the flows that carry messages sent by a source node to all the other nodes in the network. Therefore, $B_{\text{source-sw}}$ is $\frac{b_0}{p \cdot d_1}$. The switch-destination link is shared by the flows that carry messages to a destination node from all the other nodes in the network. Thus, $B_{\text{sw-destination}}$ is $\frac{b_0}{p \cdot d_1}$. Consequently, $B_1 = \min(\frac{b_0}{p \cdot d_1}, \frac{b_0}{p \cdot d_1}) = \frac{b_0}{p \cdot d_1}$.

We analyze the switch-switch link contention for the two cases when $d_1 \bmod 2 = 1$ and $d_1 \bmod 2 = 0$, separately. Due to the topology and communication pattern symmetries, the switch$_k$-switch$_{(k+1 \bmod d_1)}$ link contention is the same for all values $k \in [0, d_1 - 1]$. We quantify the number of flows that traverse the directional switch$_k$-switch$_{(k+1 \bmod d_1)}$ link.

We calculate the number of flows for the case when $d_1 \bmod 2 = 1$. The nodes attached to switch$_k$ generate p^2 flows to the nodes attached to the next $\lfloor \frac{d_1}{2} \rfloor$ switches. Due to shortest-path routing, all these flows traverse the switch$_k$-switch$_{(k+1 \bmod d_1)}$ link. Similarly the nodes attached to switch$_{(k-1) \bmod d_1}$ generate p^2 flows to the nodes attached to the next $\lfloor \frac{d_1}{2} \rfloor - 1$ switches. These flows also traverse the switch$_k$-switch$_{(k+1 \bmod d_1)}$ link. And so on until a switch$_x$ sends its p^2 flows to 1 single switch through the switch$_k$-switch$_{(k+1 \bmod d_1)}$ link.

The total number of flows that traverse the switch$_k$-switch$_{(k+1 \bmod d_1)}$ link is the sum over all the previously mentioned flows, $F = \lfloor \frac{d_1}{2} \rfloor \cdot p^2 + (\lfloor \frac{d_1}{2} \rfloor - 1) \cdot p^2 + ... + (\lfloor \frac{d_1}{2} \rfloor - (\lfloor \frac{d_1}{2} \rfloor - 1)) \cdot p^2$. If $d_1 = 2 \cdot m + 1$, then $\lfloor \frac{d_1}{2} \rfloor = m$. In this case, $F = m \cdot p^2 + (m-1) \cdot p^2 + ... + (m - (m-1)) \cdot p^2 = \frac{m \cdot (m+1)}{2} \cdot p^2 = \frac{\lfloor \frac{d_1}{2} \rfloor \cdot \lceil \frac{d_1}{2} \rceil}{2} \cdot p^2$.

We perform the same calculation for the case when $d_1 \bmod 2 = 0$. The nodes attached to switch$_k$ generate p^2 flows to the nodes attached to the next $\lfloor \frac{d_1}{2} \rfloor - 1$ switches and $0.5 \cdot p^2$ flows to the switch at $\frac{d_1}{2}$ hops distance. The 0.5 factor comes from the routing assumption that when there are two equal paths between a source a destination, like in this case when the ring dimension is an even number, the flows are equally distributed between the paths. This is equivalent with saying that switch$_k$ generates p^2 flows to the nodes attached to the next $\lfloor\frac{d_1}{2}\rfloor - 0.5$ switches. All these flows traverse the switch$_k$-switch$_{(k+1 \bmod d_1)}$ link. Similarly, the nodes attached to switch$_{(k-1) \bmod d_1}$ generate p^2 flows to the nodes attached to the next $\lfloor\frac{d_1}{2}\rfloor - 0.5 - 1$ switches. These flows also traverse the switch$_k$-switch$_{(k+1 \bmod d_1)}$ link. And so on until a switch$_x$ sends its p^2 flows to 0.5 switches through the switch$_k$-switch$_{(k+1 \bmod d_1)}$ link.

The total number of flows that traverse the switch$_k$-switch$_{(k+1 \bmod d_1)}$ link is: $F = (\lfloor\frac{d_1}{2}\rfloor - 0.5)\cdot p^2 + (\lfloor\frac{d_1}{2}\rfloor - 0.5 - 1)\cdot p^2 + ... + (\lfloor\frac{d_1}{2}\rfloor - (\lfloor\frac{d_1}{2}\rfloor - 0.5 + 1))\cdot p^2$. If $d_1 = 2\cdot m$, then $\lfloor\frac{d_1}{2}\rfloor = m$. In this case, $F = (m - 0.5)\cdot p^2 + (m - 0.5 - 1)\cdot p^2 + ... + (m - 0.5 - (m - 0.5 + 1))\cdot p^2 = \frac{m\cdot m}{2}\cdot p^2 = \frac{\lfloor\frac{d_1}{2}\rfloor\cdot\lceil\frac{d_1}{2}\rceil}{2}\cdot p^2$.

The quantity $\frac{\lfloor\frac{d_1}{2}\rfloor\cdot\lceil\frac{d_1}{2}\rceil}{2}$ corresponds to the total number of pairs of switches whose nodes communicate through the switch$_k$-switch$_{(k+1 \bmod d_1)}$ link (the two switches communicate through p^2 flows). Or, in general, in a 1D torus with d_1 nodes, the total number of pairs of switches that communicate through any link of the torus is:

$$N_{d_1} = \frac{\lfloor\frac{d_1}{2}\rfloor\cdot\lceil\frac{d_1}{2}\rceil}{2}. \tag{5.23}$$

In summary, $B_{\text{sw}-\text{sw}} = \frac{b_1}{\frac{\lfloor\frac{d_1}{2}\rfloor\cdot\lceil\frac{d_1}{2}\rceil}{2}\cdot p^2}$ and $B_2 = \min(\frac{b_0}{p\cdot d_1}, \frac{b_1}{\frac{\lfloor\frac{d_1}{2}\rfloor\cdot\lceil\frac{d_1}{2}\rceil}{2}\cdot p^2})$.

Given B_1 and B_2, the model for the effective application bandwidth in Equation 5.22 is equivalent to the model in Equation 5.24.

$$B_{\text{app}}^{\text{eff}} = (p\cdot d_1)^2\cdot\min(B_1, B_2) = p\cdot d_1\cdot\min(b_0, \frac{2\cdot b_1\cdot d_1}{\lfloor\frac{d_1}{2}\rfloor\cdot\lceil\frac{d_1}{2}\rceil\cdot p}) \tag{5.24}$$

To determine the effective bandwidth per node for the uniform communication pattern, in a 1D torus topology with dimension-order routing and linear mapping, we use the model shown in Equation 5.25, where the parameters are explained in Table 5.4.

$$B_{\text{node}}^{\text{eff}} = \frac{B_{\text{app}}^{\text{eff}}}{p\cdot d_1} = \min(b_0, \frac{2\cdot b_1\cdot d_1}{\lfloor\frac{d_1}{2}\rfloor\cdot\lceil\frac{d_1}{2}\rceil\cdot p}) \tag{5.25}$$

5.4.5 2-Dimensional Torus Topology

The 2-dimensional (2D) torus is a 2-dimensional grid with dimension-order routing scheme. Similarly to the 1D torus, if the size of the X dimension, say d_1, is an even number, then for every switch S in an X row there are two paths of equal length to access the switch at $\frac{d_1}{2}$ hops distance from switch S in the same X row. In this case, we will assume that the flows originating in S with destination nodes attached to the switch at $\frac{d_1}{2}$ hops away from S in the same X row with S are equally distributed between the two paths. We make the same assumption for flows originating in switches located in the same Y column.

The total number of compute nodes in a 2D torus is $p \cdot d_1 \cdot d_2$. The number of flows per compute node is thus $p \cdot d_1 \cdot d_2$ and the total number of flows generated by the application through the network is $(p \cdot d_1 \cdot d_2)^2$.

For the uniform pattern run on a 2D torus we identify four flow types traversing the network. The first flow type includes flows between communicating nodes that are connected to the same switch. The second and third flow types include flows between nodes connected to switches in the same X row and in the same Y column, respectively. Finally, the forth flow type includes flows between nodes connected to switches located on different X rows and Y columns.

The flows of the first type send messages through two links: source-switch and switch-destination. The source-switch link is shared by all the flows that carry messages sent by a source node to all the other nodes in the network. Therefore, $B_{\text{source-sw}}$ is $\frac{b_0}{p \cdot d_1 \cdot d_2}$. The switch-destination link is shared by the flows that carry messages to a destination node from all the other nodes in the network, thus $B_{\text{sw-destination}}$ is $\frac{b_0}{p \cdot d_1 \cdot d_2}$. The effective bandwidth of this first flow type is calculated as the minimum between the effective bandwidths achieved on each of the links:

$$B_1 = \min(B_{\text{source-sw}}, B_{\text{sw-destination}}) = \min(\frac{b_0}{p \cdot d_1 \cdot d_2}, \frac{b_0}{p \cdot d_1 \cdot d_2}) = \frac{b_0}{p \cdot d_1 \cdot d_2}. \tag{5.26}$$

The flows of the second type send messages through at least three links: source-switch, switch-switch (same X row) and switch-destination. Depending on the locations of the source and destination nodes in the network, the flows traverse one or more consecutive switch-switch links. However, due to the topology and communication pattern symmetries, the contention on the switch-switch links (same X row) is the same. We use Figure 5.9 to calculate the contention of a particular switch-switch link in the X dimension. The wrap-around links have been omitted for simplicity of drawing. Let's quantify the number of flows traversing the directional link from switch$_1$ to switch$_2$ of row R. Similar to the 1D torus case, the link is traversed by the flows of $\frac{\lfloor \frac{d_1}{2} \rfloor \cdot \lceil \frac{d_1}{2} \rceil}{2}$ pairs of switches in row R (Equation 5.23). However, in this 2D case, when a switch S (e.g., switch$_0$ of row R) communicates with another switch S' in its row

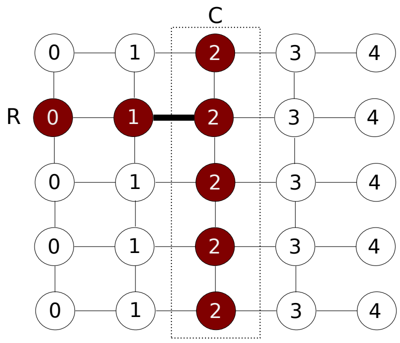


Figure 5.9 – 2D torus switch-switch (same row R) X link contention.

(e.g., switch$_2$ of row R) through a link in the X dimension (e.g., the link from switch$_0$ to switch$_2$ of row R), it also communicates through the same link with all the switches in the column of S' (the column C). Thus, in the same way as we had N_{d_1} pairs of switches communicating through any given link of a 1D torus (Equation 5.23), in the 2D case, any link in the X dimension is traversed by the flows of N_{d_1} switch-column pairs. The total number of flows between one source switch and one destination column of switches is $p^2 \cdot d_2$. Consequently, the total number of flows traversing any X link is $\frac{\lfloor\frac{d_1}{2}\rfloor\cdot\lceil\frac{d_1}{2}\rceil}{2} \cdot p^2 \cdot d_2$ and the effective bandwidth of the second flow type can be calculated as:

$$B_2 = \min(B_{\text{source-sw}}, B_{\text{sw}_X - \text{sw}_X}, B_{\text{sw-destination}}) = \min\left(\frac{b_0}{p \cdot d_1 \cdot d_2}, \frac{b_1}{\frac{\lfloor\frac{d_1}{2}\rfloor\cdot\lceil\frac{d_1}{2}\rceil}{2} \cdot p^2 \cdot d_2}\right). \tag{5.27}$$

The flows of the third flow type send messages through three types of links: source-switch, switch-switch (same Y column) and switch-destination. Depending on the locations of the source and destination nodes, the flows will traverse one or more consecutive switch-switch links. However, due to the topology and communication pattern symmetries, the contention on the switch-switch links (same Y column) are the same. We use Figure 5.10 to show how we calculate the contention of a particular switch-switch link in the Y dimension. The wrap-around links have been omitted for simplicity of drawing. Let's quantify the number of flows traversing the directional link from switch$_2$ of row R to switch$_2$ of row $R-1$. Similar to the 1D torus case, the link is traversed by the flows of $\frac{\lfloor\frac{d_2}{2}\rfloor\cdot\lceil\frac{d_2}{2}\rceil}{2}$ pairs of switches in column C (Equation 5.23). However, when a switch S (e.g., switch$_2$ of row R) communicates with another switch S' in its column (e.g., switch$_2$ of row $R-2$) through a link in the Y dimension (e.g., the link from switch$_2$ of row R to switch$_2$ of row $R-1$), all the switches in the row R of switch S also communicate with S' through the same link. Thus, in the same way as we had N_{d_1} pairs of switches communicating through any given link of a 1D torus (Equation 5.23), in the 2D case, any link in the Y dimension is traversed by the flows of N_{d_2} row-switch pairs. The total number of flows between one source row of switches and one destination switch is $p^2 \cdot d_1$. Consequently, the total number of flows traversing any Y link is $\frac{\lfloor\frac{d_2}{2}\rfloor\cdot\lceil\frac{d_2}{2}\rceil}{2} \cdot p^2 \cdot d_1$ and

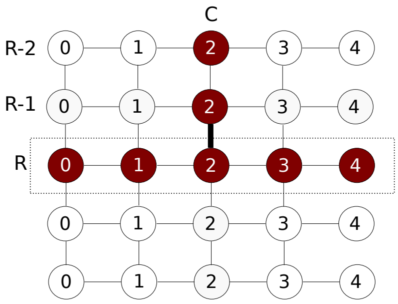


Figure 5.10 – 2D torus switch-switch (same column C) Y link contention.

the effective bandwidth of the second flow type can be calculated as:

$$B_3 = \min(B_{\text{source-sw}}, B_{\text{sw}_\text{Y}-\text{sw}_\text{Y}}, B_{\text{sw-destination}}) = \min\left(\frac{b_0}{p \cdot d_1 \cdot d_2}, \frac{b_2}{\frac{\lfloor \frac{d_2}{2} \rfloor \cdot \lceil \frac{d_2}{2} \rceil}{2} \cdot p^2 \cdot d_1}\right). \tag{5.28}$$

The flows of the forth flow type send messages through four types of links: source-switch, switch-switch (same X row), switch-switch (same Y column) and switch-destination. We have previously quantified the flow bottlenecks of the switch-switch links for switches in the same X row and for switches in the same Y column. Thus, the effective bandwidth of the forth flow type is calculated as:

$$\begin{aligned} B_4 &= \min(B_{\text{source-sw}}, B_{\text{sw}_\text{X}-\text{sw}_\text{X}}, B_{\text{sw}_\text{Y}-\text{sw}_\text{Y}}, B_{\text{sw-destination}}) \\ &= \min\left(\frac{b_0}{p \cdot d_1 \cdot d_2}, \frac{b_1}{\frac{\lfloor \frac{d_1}{2} \rfloor \cdot \lceil \frac{d_1}{2} \rceil}{2} \cdot p^2 \cdot d_2}, \frac{b_2}{\frac{\lfloor \frac{d_2}{2} \rfloor \cdot \lceil \frac{d_2}{2} \rceil}{2} \cdot p^2 \cdot d_1}\right). \end{aligned} \tag{5.29}$$

By applying Equation 5.2, we calculate the application bandwidth as in Equation 5.30.

$$\begin{aligned} B_{\text{app}}^{\text{eff}} &= \frac{(p \cdot d_1 \cdot d_2)^2 \cdot V}{\max(\frac{V}{B_1}, \frac{V}{B_2}, \frac{V}{B_3}, \frac{V}{B_4})} = (p \cdot d_1 \cdot d_2)^2 \cdot \min(B_1, B_2, B_3, B_4) \\ &= (p \cdot d_1 \cdot d_2)^2 \cdot \min\left(\frac{b_0}{p \cdot d_1 \cdot d_2}, \frac{b_1}{\frac{\lfloor \frac{d_1}{2} \rfloor \cdot \lceil \frac{d_1}{2} \rceil}{2} \cdot p^2 \cdot d_2}, \frac{b_2}{\frac{\lfloor \frac{d_2}{2} \rfloor \cdot \lceil \frac{d_2}{2} \rceil}{2} \cdot p^2 \cdot d_1}\right) \end{aligned} \tag{5.30}$$

To determine the effective bandwidth per node for the uniform communication pattern, in a 2D torus with dimension-order routing and linear mapping, we use the model shown in Equation 5.31, where the parameters are explained in Table 5.4.

$$B_{\text{node}}^{\text{eff}} = \frac{B_{\text{app}}^{\text{eff}}}{p \cdot d_1 \cdot d_2} = \min\left(b_0, \frac{2 \cdot b_1 \cdot d_1}{\lfloor \frac{d_1}{2} \rfloor \cdot \lceil \frac{d_1}{2} \rceil \cdot p}, \frac{2 \cdot b_2 \cdot d_2}{\lfloor \frac{d_2}{2} \rfloor \cdot \lceil \frac{d_2}{2} \rceil \cdot p}\right) \tag{5.31}$$

5.4.6 3-Dimensional Torus Topology

The total number of compute nodes in the 3-dimensional (3D) torus topology is $p \cdot d_1 \cdot d_2 \cdot p_3$. For the uniform communication pattern, the number of flows per compute node is thus $p \cdot d_1 \cdot d_2 \cdot p_3$ and the total number of flows generated by the application through the network is $(p \cdot d_1 \cdot d_2 \cdot p_3)^2$.

Similarly to the 1D and 2D torus cases, if the size of say the X dimension d_1 is an even number, then for every switch S in an X row there are two paths of equal length to access the switch at $\frac{d_1}{2}$ hops away from switch S in the same X row. In this case, we will assume that the flows originating in S with destination nodes attached to the switch at $\frac{d_1}{2}$ hops away from S in the same X row with S are equally distributed between the two paths. We make the same assumption for communicating flows through switches of the same Y or Z rows.

For the uniform pattern on the 3D torus we identify eight flow types traversing the network. The first flow type includes flows between communicating nodes that are connected to the same switch. The second, third and forth flow types include flows between nodes connected to switches in the same X, same Y and same Z row, respectively. The fifth, sixth and seventh flow types include flows between nodes in the same XY plane, same YZ and same XZ planes, respectively. The last flow type includes flows between nodes connected to switches in different 2D planes.

For each of these flow types, the effective bandwidth is given by the minimum across the bandwidth bottlenecks imposed by each type of link that they traverse. For reasons of symmetry of topology and communication pattern, all links in the same dimension exhibit the same bottleneck. It suffices to quantify the five bottlenecks (node-switch, switch-switch in the X dimension, switch-switch in the Y dimension, switch-switch in the Z dimension and switch-node) to determine the effective bandwidths of the eight types of flows. Similarly to the 1D and 2D torus cases, the following can be shown.

1. The node-switch and switch-node links have an effective bandwidth of $\frac{b_0}{p \cdot d_1 \cdot d_2 \cdot d_3}$.
2. The switch-switch links in the X dimension are traversed by flows between communicating pairs of source switches and destination YZ planes of switches, each plane corresponding to a single switch in the same X row as the source switch. There are N_{d_1} such pairs (Eq. 5.23) and every pair has $p^2 \cdot d2 \cdot d3$ flows. Thus, the effective bandwidth on switch-switch X links is $B_{\mathrm{sw_X - sw_X}} = \frac{b_1}{p^2 \cdot d2 \cdot d3 \cdot \frac{\lfloor \frac{d_1}{2} \rfloor \cdot \lceil \frac{d_1}{2} \rceil}{2}}$.
3. The switch-switch links in the Y dimension are traversed by flows between communicating pairs of source rows of switches in the X dimension and destination rows of switches in the Z dimension, each X and Z rows corresponding to a single switch in the Y row of the link. Thus, the number of communication row-row pairs is N_{d_2} (Equation 5.23) and every pair has $p^2 \cdot d1 \cdot d3$ flows. Thus, the effective bandwidth on switch-switch Y links is $B_{\mathrm{sw_Y - sw_Y}} = \frac{b_2}{p^2 \cdot d1 \cdot d3 \cdot \frac{\lfloor \frac{d_2}{2} \rfloor \cdot \lceil \frac{d_2}{2} \rceil}{2}}$.
4. The switch-switch links in the Z dimension are traversed by flows between communicating pairs of source XY planes of switches and destination switches in the Z dimension, each XY plane corresponding to a single switch in the same Z row as the destination switch. There are N_{d_3} (Equation 5.23) such pairs and every pair has $p^2 \cdot d1 \cdot d2$ flows. Thus, the effective bandwidth on switch-switch Z links is $B_{\mathrm{sw_Z - sw_Z}} = \frac{b_3}{p^2 \cdot d1 \cdot d2 \cdot \frac{\lfloor \frac{d_3}{2} \rfloor \cdot \lceil \frac{d_3}{2} \rceil}{2}}$.

The effective bandwidth of each of the flow types B_i $i \in [1,8]$ is expressed as the minimum of a combination of the five link bottlenecks enumerated above. By applying Equation 5.2, we calculate the application bandwidth as in Equation 5.32.

$$B_{\text{app}}^{\text{eff}} = \frac{(p \cdot d_1 \cdot d_2 \cdot d_3)^2 \cdot V}{\max\limits_{1 \le i \le 8}(\frac{V}{B_i})} = (p \cdot d_1 \cdot d_2 \cdot d_3)^2 \cdot \min\limits_{1 \le i \le 8}(B_i) = (p \cdot d_1 \cdot d_2 \cdot d_3)^2 \cdot$$
$$\min(\frac{b_0}{p \cdot d_1 \cdot d_2 \cdot d_3}, \frac{b_1}{p^2 \cdot d2 \cdot d3 \cdot \frac{\lfloor \frac{d_1}{2} \rfloor \cdot \lceil \frac{d_1}{2} \rceil}{2}}, \frac{b_2}{p^2 \cdot d1 \cdot d3 \cdot \frac{\lfloor \frac{d_2}{2} \rfloor \cdot \lceil \frac{d_2}{2} \rceil}{2}}, \frac{b_3}{p^2 \cdot d1 \cdot d2 \cdot \frac{\lfloor \frac{d_3}{2} \rfloor \cdot \lceil \frac{d_3}{2} \rceil}{2}}) \tag{5.32}$$

To determine the effective bandwidth per node for the uniform communication pattern, in a 3D torus with dimension-order routing and linear mapping, we use the model shown in Equation 5.33, where the parameters are explained in Table 5.4.

$$B_{\text{node}}^{\text{eff}} = \frac{B_{\text{app}}^{\text{eff}}}{p \cdot d_1 \cdot d_2 \cdot d_3} = \min(b_0, \frac{2 \cdot b_1 \cdot d_1}{p \cdot \lfloor \frac{d_1}{2} \rfloor \cdot \lceil \frac{d_1}{2} \rceil}, \frac{2 \cdot b_2 \cdot d_2}{p \cdot \lfloor \frac{d_2}{2} \rfloor \cdot \lceil \frac{d_2}{2} \rceil}, \frac{2 \cdot b_3 \cdot d_3}{p \cdot \lfloor \frac{d_3}{2} \rfloor \cdot \lceil \frac{d_3}{2} \rceil}) \tag{5.33}$$

Based on the models derived for the 1D, 2D and 3D tori for the effective bandwidth per node of a uniform communication pattern, we can generalize the model to an N-dimensional torus as in Equation 5.34.

$$B_{\text{node}}^{\text{eff}} = \min(b_0, \min\limits_{1 \le i \le n}(\frac{2 \cdot b_i \cdot d_i}{p \cdot \lfloor \frac{d_i}{2} \rfloor \cdot \lceil \frac{d_i}{2} \rceil})) \tag{5.34}$$

5.4.7 2-Dimensional HyperX Topology

The total number of compute nodes in the 2D HyperX topology is $p \cdot d_1 \cdot d_2$. For the uniform pattern, the number of flows per compute node is $p \cdot d_1 \cdot d_2$. Therefore, the total number of flows generated by the application is $(p \cdot d_1 \cdot d_2)^2$.

For the uniform pattern run on a 2D HyperX we identify four flow types. The first flow type includes flows between communicating nodes that are connected to the same switch. The second and the third flow type include flows between nodes connected to different switches, but both switches in the same X row or Y column, respectively. The forth flow type includes flows between nodes connected to switches in different rows and columns.

Flows of the first type send messages through two links: source-switch and switch-destination. The effective bandwidth of this first flow type is calculated as the minimum between the two effective bandwidths achieved on each of the links:

$$B_1 = \min(B_{\text{source-sw}}, B_{\text{sw-destination}}). \tag{5.35}$$

Flows of the second flow type send the messages through at least three links: source-switch, switch-switch (X dimension) and switch-destination. Due to symmetries of the network topology and communication pattern, the switch-switch X links exhibit the same contention (which we calculate later in this section). Thus, the effective bandwidth of the second flow type is calculated as:

$$B_2 = \min(B_{\text{source-sw}}, B_{\text{sw}_\text{X}-\text{sw}_\text{X}}, B_{\text{sw-destination}}). \tag{5.36}$$

Similarly, flows of the third flow type send the messages also through at least three links, source-switch, switch-switch (Y dimension) and switch-destination and their effective bandwidth is calculated as:

$$B_3 = \min(B_{\text{source-sw}}, B_{\text{sw}_\text{Y}-\text{sw}_\text{Y}}, B_{\text{sw-destination}}). \tag{5.37}$$

Finally, flows of the forth flow type send the message through at least four links: source-switch, switch-switch (X dimension), switch-switch (Y dimension) and switch-destination. The effective bandwidth of this type of flow is:

$$B_4 = \min(B_{\text{source-sw}}, B_{\text{sw}_\text{X}-\text{sw}_\text{X}}, B_{\text{sw}_\text{Y}-\text{sw}_\text{Y}}, B_{\text{sw-destination}}). \tag{5.38}$$

By applying Equation 5.2, we calculate the application bandwidth as in Equation 5.39.

$$B_{\text{app}}^{\text{eff}} = \frac{(p \cdot d_1 \cdot d_2)^2 \cdot V}{\max(\frac{V}{B_1}, \frac{V}{B_2}, \frac{V}{B_3}, \frac{V}{B_4})} = (p \cdot d_1 \cdot d_2)^2 \cdot \min(B_1, B_2, B_3, B_4) \tag{5.39}$$

To calculate the B_i bandwidths, we analyze the contention on the four types of (directional) links: source-switch, switch-switch (X and Y links) and switch-destination. The source-switch link is shared by all the flows that carry messages sent by the source node to all the other nodes in the network. Therefore, $B_{\text{source-sw}}$ is $\frac{b_0}{p \cdot d_1 \cdot d_2}$. The switch-destination link is shared by all the flows that carry messages to the destination node from all the other nodes in the network. Therefore, $B_{\text{sw-destination}}$ is $\frac{b_0}{p \cdot d_1 \cdot d_2}$. Consequently, $B_1 = \min(\frac{b_0}{p \cdot d_1 \cdot d_2}, \frac{b_0}{p \cdot d_1 \cdot d_2}) = \frac{b_0}{p \cdot d_1 \cdot d_2}$.

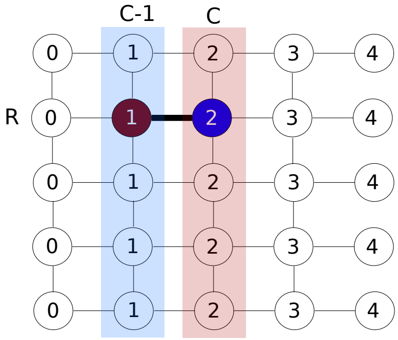


Figure 5.11 – 2D HyperX switch-switch (same X row) link contention.

Moreover, for any given link connecting two switches in the X dimension, say connecting switch S_1 to switch S_2, the link is traversed by the flows generated by the p nodes connected to S_1 towards the p nodes of S_2. However, these are not the only flows sharing this link. We use Figure 5.11 to show how we calculate the number of flows sharing the switch-switch X link.

The nodes connected to all the switches in the column $C-1$ that need to communicate with the nodes connected to switch$_2$ in column C will also send half of their flows through the same switch$_1$-switch$_2$ link of row R. They will send only half of their flows due to the routing scheme (shortest path) and due to the assumption that if there are equal paths between the source and the destination, we assume the flows to be equally distributed across the available paths. In addition switch$_1$ will send the half of its flows through the same link to nodes connected to all the switches in column C, except for switch$_2$ to which switch$_1$ sends all p^2 flows.

Thus, the total number of flows crossing the switch-switch X link is $(1+0.5\cdot(d_2-1)+0.5\cdot(d_2-1))\cdot p^2 = d_2\cdot p^2$ and the effective bandwidth B_2 is $\min(\frac{b_0}{p\cdot d_1\cdot d_2}, \frac{b_1}{p^2\cdot d_2})$. Similarly, for the switch-switch Y link the number of flows sharing a Y link is $p^2\cdot d_1$. Thus, the effective bandwidth B_3 is $\min(\frac{b_0}{p\cdot d_1\cdot d_2}, \frac{b_2}{p^2\cdot d_1})$.

Given all the link bottlenecks, we can now also calculate the effective bandwidth for the flows of the forth type and we obtain $B_4 = \min(\frac{b_0}{p\cdot d_1\cdot d_2}, \frac{b_1}{p^2\cdot d_2}, \frac{b_2}{p^2\cdot d_1})$.

Given B_1, B_2, B_3 and B_4, the effective application bandwidth shown in Equation 5.39 is equivalent to the model shown in Equation 5.40.

$$B_{\text{app}}^{\text{eff}} = p\cdot d_1\cdot d_2\cdot\min(b_0, \frac{b_1\cdot d_1}{p}, \frac{b_2\cdot d_2}{p}) \tag{5.40}$$

To determine the effective bandwidth per node for the uniform communication pattern, in a 2D HyperX topology with shortest-path routing and linear mapping, we use the model shown in Equation 5.41, where the parameters are explained in Table 5.3.

$$B_{\text{node}}^{\text{eff}} = \frac{B_{\text{app}}^{\text{eff}}}{p\cdot d_1\cdot d_2} = \min(b_0, \frac{b_1\cdot d_1}{p}, \frac{b_2\cdot d_2}{p}) \tag{5.41}$$

5.5 Bandwidth Models: Shift Communication Pattern

In a shift communication pattern, each MPI process communicates only with one other process. If the application has n processes, the process with index a sends data to the process with index $(a+s)$ mod n, where s is the shift step. Thus, each node generates only a single flow of data throughout the network. Each of these flows transports the same volume of data V.

5.5.1 Full-Mesh Topology

As the total number of compute nodes in a full-mesh topology is $a\cdot p$, the total number of flows generated by the application through the network is $a\cdot p$. The number of application flow types depends on the value of the shift step s.

We will first calculate the effective bandwidth for the case when the shift step is lower than the number of nodes connected to a switch $s<p$. There are two types of flows generated

throughout the network. The first flow type sends messages through two links: source-switch and switch-destination. There are $a \cdot (p - s)$ flows of this type, as in each switch there are $p - s$ nodes that communicate with nodes connected to the same switch. The effective bandwidth of this first flow type is calculated as the minimum between the two effective bandwidths achieved on each of the links:

$$B_1 = \min(B_{\text{source-sw}}, B_{\text{sw-destination}}). \tag{5.42}$$

The second flow type sends the messages through three links: source-switch, switch-switch and switch-destination. There are $a \cdot s$ flows that are of this type, as in each switch there are s nodes that communicate with nodes outside the switch to which they are connected. In this case, the effective bandwidth is calculated as:

$$B_2 = \min(B_{\text{source-sw}}, B_{\text{sw}-\text{sw}}, B_{\text{sw-destination}}). \tag{5.43}$$

By applying Equation 5.2, we calculate the application bandwidth as in Equation 5.44.

$$B_{\text{app}}^{\text{eff}} = \frac{a \cdot (p - s) \cdot V + a \cdot s \cdot V}{\max(\frac{V}{B_1}, \frac{V}{B_2})} = a \cdot p \cdot \min(B_1, B_2) \tag{5.44}$$

To calculate B_1 and B_2 we analyze the contention on the three types of (directional) links: source-switch, switch-switch and switch-destination. The source-switch link is used by a single flow that carries messages sent by the source node to its destination node. Therefore, $B_{\text{source}-\text{sw}}$ is the full available bandwidth b_0. This is also valid for the switch-destination link which is used by the flow that carries messages to the destination node from another single node in the network. Consequently, $B_1 = b_0$. The switch-to-switch link is shared by s flows, as s of the p nodes connected to a switch need to send data to s nodes connected to another same switch. Therefore, $B_{\text{sw}-\text{sw}} = \frac{b_1}{s}$ and $B_2 = \min(b_0, \frac{b_1}{s})$. Equation 5.45 shows the effective application bandwidth for $s < p$.

$$B_{\text{app}}^{\text{eff}} = a \cdot p \cdot \min(b_0, \min(b_0, \frac{b_1}{s})) = a \cdot p \cdot \min(b_0, \frac{b_1}{s}) \tag{5.45}$$

We analyze the case when the shift step is greater than the number of nodes connected to a switch $s > p$ and $s \bmod p \neq 0$. For any given switch S_1 the p flows originating in its nodes will go each to one of two switches S_2 and S_3. Specifically $s' = s \bmod p$ of the flows go to one of the switches, say S_2, and the remaining $p - s'$ flows go to the other switch. For all of these flows their effective bandwidth will be:

$$B = \min(B_{\text{source-sw}}, B_{\text{sw}-\text{sw}}, B_{\text{sw-destination}}). \tag{5.46}$$

As before, $B_{\text{source-sw}}$ and $B_{\text{sw-destination}}$ will be b_0. The s' flows from S_1 to S_2 share the link connecting S_1 to S_2. As we are using shortest path routing in a full-mesh, no other network flows use this link. Thus, $B_{\text{sw}-\text{sw}}$ is $\frac{b_1}{s'}$ for these flows. Similarly, for the flows from S_1 to S_3,

$B_{\text{sw}-\text{sw}}$ is $\frac{b_1}{p-s'}$. Consequently, we have two flow types, one with $B_1 = \min(b_0, \frac{b_1}{s'})$ and one with $B_2 = \min(b_0, \frac{b_1}{p-s'})$. The effective bandwidth of the application will be:

$$B_{\text{app}}^{\text{eff}} = \frac{a \cdot s' \cdot V + a \cdot (p - s') \cdot V}{\max(\frac{V}{B_1}, \frac{V}{B_2})} = a \cdot p \cdot \min(B_1, B_2) = a \cdot p \cdot \min(b_0, \frac{b_1}{s'}, \frac{b_1}{p - s'}) \tag{5.47}$$

We identify a corner case in which the shift step is a multiple of the number of nodes connected to a switch $s' = s \bmod p = 0$. For any given switch S_1 the p flows originating in its nodes will go each to only one other switch S_2. For all these flows their effective bandwidth will be:

$$B_1 = \min(B_{\text{source-sw}}, B_{\text{sw}-\text{sw}}, B_{\text{sw-destination}}). \tag{5.48}$$

As before, $B_{\text{source-sw}}$ and $B_{\text{sw-destination}}$ will be b_0. The p flows from S_1 to S_2 share the link connecting S_1 to S_2. As we are using shortest path routing in a full-mesh, no other network flows use this link. Thus, $B_{\text{sw}-\text{sw}}$ is $\frac{b_1}{p}$ for these flows. Consequently, $B_1 = \min(b_0, \frac{b_1}{p})$. The effective bandwidth of the application will be:

$$B_{\text{app}}^{\text{eff}} = \frac{a \cdot p \cdot V}{\frac{V}{B_1}} = a \cdot p \cdot \min(B_1) = a \cdot p \cdot \min(b_0, \frac{b_1}{p}) \tag{5.49}$$

In summary, to determine the effective bandwidth per node for the shift communication pattern, in a full-mesh topology with shortest-path routing and linear mapping, we use the model in Equation 5.50, where $s' = s \bmod p$. The hardware parameters are explained in Table 5.1.

$$B_{\text{node}}^{\text{eff}} = \frac{B_{\text{app}}^{\text{eff}}}{a \cdot p} = \begin{cases} \min(b_0, \frac{b_1}{s}) & s < p \\ \min(b_0, \frac{b_1}{p}) & s' = 0 \\ \min(b_0, \frac{b_1}{s'}, \frac{b_1}{p-s'}) & s > p \land s' \neq 0 \end{cases} \tag{5.50}$$

5.5.2 2-Level Fat-Tree Topology

The total number of compute nodes in a 2-level fat-tree topology is $m_1 \cdot m_2$ and, in the case of the shift communication pattern, the total number of flows generated by the application through the network is $m_1 \cdot m_2$. The effective bandwidth per application depends on the value of the shift step s.

We will first calculate the effective bandwidth for the case when the shift step is lower than the number of nodes connected to a switch$_{\text{L1}}$ $s < m_1$. For each switch$_{\text{L1}}$ in the network, $m_1 - s$ nodes connected to switch$_{\text{L1}}$ will each send one flow to $m_1 - s$ nodes connected to the same switch$_{\text{L1}}$. These flows will traverse two links, source-switch$_{\text{L1}}$ and switch$_{\text{L1}}$-destination. The source-switch$_{\text{L1}}$ link is used by a single flow that carries messages sent by the source node to its destination node. This is also valid for the switch$_{\text{L1}}$-destination link which is used by the flow that carries messages to the destination node from another single node in the network. Thus, the effective bandwidth obtained by the $m_1 - s$ flows is $B_1 = \min(B_{\text{source}-\text{sw}_{\text{L1}}}, B_{\text{sw}_{\text{L1}}-\text{destination}}) = \min(b_0, b_0) = b_0$.

The remaining s nodes will generate s flows that will exit their corresponding switch$_{\text{L1}}$. These flows will traverse four links, source-switch$_{\text{L1}}$, switch$_{\text{L1}}$-switch$_{\text{L2}}$, switch$_{\text{L2}}$-switch$_{\text{L1}}$ and switch$_{\text{L1}}$-destination. The effective bandwidth of these flows is calculated as the minimum across the effective bandwidths achieved on the four links:

$$B_2 = \min(B_{\text{source-sw}_{\text{L1}}}, B_{\text{sw}_{\text{L1}}-\text{sw}_{\text{L2}}}, B_{\text{sw}_{\text{L2}}-\text{sw}_{\text{L1}}}, B_{\text{sw}_{\text{L1}}-\text{destination}}). \tag{5.51}$$

For a given switch$_{\text{L1}}$, the s flows that exit the switch will share the w_1 switch$_{\text{L1}}$-switch$_{\text{L2}}$ links, thus each flow achieving an effective bandwidth of $B_{\text{sw}_{\text{L1}}-\text{sw}_{\text{L2}}} = \frac{b_1 \cdot w_1}{s}$. Moreover, for each switch$_{\text{L1}}$, there are exactly s flows sent from another L1 switch to s nodes connected to switch$_{\text{L1}}$. These flows traverse w_1 switch$_{\text{L2}}$-switch$_{\text{L1}}$ links. Therefore, the effective bandwidth obtained on the switch$_{\text{L2}}$-switch$_{\text{L1}}$ links is also $B_{\text{sw}_{\text{L2}}-\text{sw}_{\text{L1}}} = \frac{b_1 \cdot w_1}{s}$. In conclusion, the s flows that exit switch$_{\text{L1}}$ achieve an effective bandwidth of $B_2 = \min(b_0, \frac{b_1 \cdot w_1}{s}, \frac{b_1 \cdot w_1}{s}, b_0) = \min(b_0, \frac{b_1 \cdot w_1}{s})$.

By applying Equation 5.2, we calculate the application bandwidth for $s < m_1$ as:

$$B_{\text{app}}^{\text{eff}} = \frac{m_2 \cdot (m_1 - s) \cdot V + m_2 \cdot s \cdot V}{\max(\frac{V}{B_1}, \frac{V}{B_2})} = m_1 \cdot m_2 \cdot \min(b_0, \frac{b_1 \cdot w_1}{s}). \tag{5.52}$$

For $s \geq m_1$, the m_1 nodes connected to a switch$_{\text{L1}}$ send all their m_1 flows to nodes connected to another switch$_{\text{L1}}$. Thus, all flows traverse the network through four links: source-switch$_{\text{L1}}$, switch$_{\text{L1}}$-switch$_{\text{L2}}$, switch$_{\text{L2}}$-switch$_{\text{L1}}$ and switch$_{\text{L1}}$-destination. All m_1 flows exit their corresponding switch$_{\text{L1}}$ and share w_1 switch$_{\text{L1}}$-switch$_{\text{L2}}$ links. Thus, the effective bandwidth on the switch$_{\text{L1}}$-switch$_{\text{L2}}$ links is $\frac{b_1 \cdot w_1}{s}$. Due to network and pattern symmetries, the same effective bandwidth is obtained on the switch$_{\text{L2}}$-switch$_{\text{L1}}$ links. In summary, the effective bandwidth of the m_1 flows is $B_3 = \min(b_0, \frac{b_1 \cdot w_1}{m_1})$.

By applying Equation 5.2, we calculate the application bandwidth for $s \geq m_1$ as:

$$B_{\text{app}}^{\text{eff}} = \frac{m_2 \cdot m_1 \cdot V}{\max(\frac{V}{B_3})} = m_1 \cdot m_2 \cdot \min(b_0, \frac{b_1 \cdot w_1}{m_1}). \tag{5.53}$$

In summary, to determine the effective bandwidth per node for the shift communication pattern, in a 2-level fat-tree topology with shortest-path routing and linear mapping, we use the model in Equation 5.54. The hardware parameters are explained in Table 5.2.

$$B_{\text{node}}^{\text{eff}} = \frac{B_{\text{app}}^{\text{eff}}}{m_1 \cdot m_2} = \begin{cases} \min(b_0, \frac{b_1 \cdot w_1}{s}) & s < m_1 \\ \min(b_0, \frac{b_1 \cdot w_1}{m_1}) & s \geq m_1 \end{cases} \tag{5.54}$$

5.5.3 3-Level Fat-Tree Topology

The total number of compute nodes in a 3-level fat-tree topology is $m_1 \cdot m_2 \cdot m_3$ and the total number of flows generated by the application through the network is $m_1 \cdot m_2 \cdot m_3$. The effective bandwidth per application depends on the value of the shift step s.

We will first calculate the effective bandwidth for the case when the shift step is lower than the number of nodes connected to a switch$_{\mathrm{L1}}$ $s < m_1$. Let's analyze the flows in a switch$_{\mathrm{L2}}$-rooted sub-tree. The same analysis will apply to all the m_3 sub-trees in the network.

For each of the first $m_2 - 1$ switches switch$_{\mathrm{L1}}$ in the sub-tree, $m_1 - s$ nodes connected to switch$_{\mathrm{L1}}$ send their flows to $m_1 - s$ nodes connected to the same switch switch$_{\mathrm{L1}}$. Thus, the flows traverse two links, source-switch$_{\mathrm{L1}}$ and switch$_{\mathrm{L1}}$-destination. The source-switch$_{\mathrm{L1}}$ link is used by a single flow that carries messages sent by the source node to its destination node. Therefore, the effective bandwidth on the source-switch$_{\mathrm{L1}}$ link is the full available bandwidth b_0. This is also valid for the switch$_{\mathrm{L1}}$-destination link which is used by the flow that carries messages to the destination node from another single node in the network. Thus, the effective bandwidth obtained by the $m_1 - s$ flows is $B_1 = \min(b_0, b_0) = b_0$.

The remaining s nodes of the $m_2 - 1$ switches generate s flows that exit their corresponding switch$_{\mathrm{L1}}$. These flows traverse four links, source-switch$_{\mathrm{L1}}$, switch$_{\mathrm{L1}}$-switch$_{\mathrm{L2}}$, switch$_{\mathrm{L2}}$-switch$_{\mathrm{L1}}$ and switch$_{\mathrm{L1}}$-destination. The effective bandwidth of these s flows is calculated as the minimum across the effective bandwidths achieved on the four links:

$$B_2 = \min(B_{\text{source-sw}_{\mathrm{L1}}}, B_{\mathrm{sw}_{\mathrm{L1}}-\mathrm{sw}_{\mathrm{L2}}}, B_{\mathrm{sw}_{\mathrm{L2}}-\mathrm{sw}_{\mathrm{L1}}}, B_{\mathrm{sw}_{\mathrm{L1}}-\text{destination}}). \tag{5.55}$$

For a given switch$_{\mathrm{L1}}$, the s flows that exit the switch share the w_1 switch$_{\mathrm{L1}}$-switch$_{\mathrm{L2}}$ links, thus each flow achieving an effective bandwidth of $B_{\mathrm{sw}_{\mathrm{L1}}-\mathrm{sw}_{\mathrm{L2}}} = \frac{b_1 \cdot w_1}{s}$. Moreover, for a given switch$_{\mathrm{L1}}$, there are exactly s flows sent from another switch$_{\mathrm{L1}}$ to s nodes connected to switch$_{\mathrm{L1}}$. These flows traverse w_1 switch$_{\mathrm{L2}}$-switch$_{\mathrm{L1}}$ links. Therefore, the effective bandwidth obtained on the switch$_{\mathrm{L2}}$-switch$_{\mathrm{L1}}$ links is also $B_{\mathrm{sw}_{\mathrm{L2}}-\mathrm{sw}_{\mathrm{L1}}} = \frac{b_1 \cdot w_1}{s}$. In conclusion, the s flows that exit switch$_{\mathrm{L1}}$ achieve an effective bandwidth of $B_2 = \min(b_0, \frac{b_1 \cdot w_1}{s}, \frac{b_1 \cdot w_1}{s}, b_0) = \min(b_0, \frac{b_1 \cdot w_1}{s})$.

For the last switch switch$_{\mathrm{L1}}$ in the switch$_{\mathrm{L2}}$-rooted sub-tree, $m_1 - s$ flows are sent to $m_1 - s$ nodes connected to the same switch$_{\mathrm{L1}}$ at the same $B_1 = b_0$ bandwidth. However, s flows are sent to the first switch$_{\mathrm{L1}}$ in the next sub-tree. Thus, these last s flows traverse six links in the network source-switch$_{\mathrm{L1}}$, switch$_{\mathrm{L1}}$-switch$_{\mathrm{L2}}$, switch$_{\mathrm{L2}}$-switch$_{\mathrm{L3}}$, switch$_{\mathrm{L3}}$-switch$_{\mathrm{L2}}$, switch$_{\mathrm{L2}}$-switch$_{\mathrm{L1}}$ and switch$_{\mathrm{L1}}$-destination. The s flows share the w_1 switch$_{\mathrm{L1}}$-switch$_{\mathrm{L2}}$ links and the $w_1 \cdot w_2$ switch$_{\mathrm{L2}}$-switch$_{\mathrm{L3}}$ links. Thus, upward in the tree, each of the s flows achieve an effective bandwidth of $\min(b_0, \frac{b_1 \cdot w_1}{s}, \frac{b_2 \cdot w_1 \cdot w_2}{s})$. Due to symmetries of the network and communication pattern, the same bandwidth is obtained on the downward links in the tree, switch$_{\mathrm{L3}}$-switch$_{\mathrm{L2}}$, switch$_{\mathrm{L2}}$-switch$_{\mathrm{L1}}$ and switch$_{\mathrm{L1}}$-destination. Indeed, for example, given switch$_{\mathrm{L2}}$, there are exactly s flows sent from another sub-tree to nodes in the switch$_{\mathrm{L2}}$-rooted sub-tree. Thus, the

$w_1 \cdot w_2$ switch$_{\text{L3}}$-switch$_{\text{L2}}$ links are shared by s flows. Consequently, the s flows that exit their switch$_{\text{L2}}$-rooted sub-tree achieve an effective bandwidth of $B_3 = \min(b_0, \frac{b_1 \cdot w_1}{s}, \frac{b_2 \cdot w_1 \cdot w_2}{s})$.

By applying Equation 5.2, we calculate the application bandwidth for $s < m_1$ as:

$$B_{\text{app}}^{\text{eff}} = \frac{m_3 \cdot m_2 \cdot (m_1 - s) \cdot V + m_3 \cdot (m_2 - 1) \cdot s \cdot V + m_3 \cdot s}{\max(\frac{V}{B_1}, \frac{V}{B_2}, \frac{V}{B_3})} \tag{5.56}$$

$$= m_1 \cdot m_2 \cdot m_3 \cdot \min(b_0, \frac{b_1 \cdot w_1}{s}, \frac{b_2 \cdot w_1 \cdot w_2}{s}). \tag{5.57}$$

For $m_1 \leq s < m_1 \cdot m_2$, the only difference when compared with the $s < m_1$ case is that all nodes in a switch$_{\text{L1}}$ switch send m_1 flows sharing the available w_1 switch$_{\text{L1}}$-switch$_{\text{L2}}$ links. Thus, the switch$_{\text{L1}}$-switch$_{\text{L2}}$ and the switch$_{\text{L2}}$-switch$_{\text{L1}}$ link bottlenecks are $\frac{b_1 \cdot w_1}{m_1}$. The application bandwidth for $m_1 \leq s < m_1 \cdot m_2$ will be:

$$B_{\text{app}}^{\text{eff}} = m_1 \cdot m_2 \cdot m_3 \cdot \min(b_0, \frac{b_1 \cdot w_1}{m_1}, \frac{b_2 \cdot w_1 \cdot w_2}{s}). \tag{5.58}$$

Finally, for $s \geq m_1 \cdot m_2$, the only difference when compared with the previous case is that all nodes $m_1 \cdot m_2$ in a switch$_{\text{L2}}$-rooted sub-tree send their flows outside their corresponding sub-tree, sharing the available $w_1 \cdot w_2$ switch$_{\text{L2}}$-switch$_{\text{L3}}$ links. Thus, the switch$_{\text{L2}}$-switch$_{\text{L3}}$ and switch$_{\text{L3}}$-switch$_{\text{L2}}$ link bottlenecks are $\frac{b_2 \cdot w_1 \cdot w_2}{s}$. The application bandwidth for $s \geq m_1 \cdot m_2$ will be:

$$B_{\text{app}}^{\text{eff}} = m_1 \cdot m_2 \cdot m_3 \cdot \min(b_0, \frac{b_1 \cdot w_1}{m_1}, \frac{b_2 \cdot w_1 \cdot w_2}{m_1 \cdot m_2}). \tag{5.59}$$

In summary, to determine the effective bandwidth per node for the shift communication pattern, in a 3-level fat-tree topology with shortest-path routing and linear mapping, we use the model in Equation 5.60. The hardware parameters are explained in Table 5.2.

$$B_{\text{node}}^{\text{eff}} = \frac{B_{\text{app}}^{\text{eff}}}{m_1 \cdot m_2 \cdot m_3} = \begin{cases} \min(b_0, \frac{b_1 \cdot w_1}{s}, \frac{b_2 \cdot w_1 \cdot w_2}{s}) & s < m_1 \\ \min(b_0, \frac{b_1 \cdot w_1}{m_1}, \frac{b_2 \cdot w_1 \cdot w_2}{s}) & m_1 \leq s < m_1 \cdot m_2 \\ \min(b_0, \frac{b_1 \cdot w_1}{m_1}, \frac{b_2 \cdot w_1 \cdot w_2}{m_1 \cdot m_2}) & s \geq m_1 \cdot m_2 \end{cases} \tag{5.60}$$

5.6 Bandwidth Models: Nearest-Neighbor Communication Pattern

We start this section by describing the MPI rank mapping strategies covered by our nearest-neighbor models. Then we will derive the bandwidth models for the 2-dimensional nearest-neighbor (2D NNE) pattern. In this pattern, each process communicates with 4 processes corresponding to its North/South/East/West neighbors in the application domain. Each process generates 4 flows to their neighbors, one to each neighbor, each flow carrying on average the same amount of data.

5.6.1 Overview of Supported MPI Rank Mappings

An MPI rank mapping strategy defines what hardware node is assigned to which application process. As mentioned in Section 5.1, due to its relevance in practice, for the nearest-neighbor communication pattern run on full-mesh and fat-tree topologies, we will consider more than just the linear mapping.

Specifically, we consider mappings where the application domain (the grid of processes that determines the nearest-neighbor pattern) is partitioned into equal-sized application sub-domains. Furthermore, we consider the domain of hardware nodes to be also partitioned into equal-sized hardware sub-domains. In the case of a full-mesh, a partitioning of the hardware nodes into switches is such a partition (a hardware sub-domain is a switch). The mappings that we cover are those that bijectively map the application sub-domains onto the hardware sub-domains. Thus, not only must the total number of nodes in the system match the total number of application processes, but in addition the processes in an application sub-domain should fully populate the compute nodes in a hardware sub-domain. An example of such a mapping is shown in [109] for dragonfly topologies.

Figures 5.12 and 5.13 show an example of such a mapping. In Figure 5.13 we show the hardware domain or the network topology (in this case, a full-mesh with $a = 6$ switches and $p = 10$ nodes per switch). We do not show the compute nodes to simplify the figure. In Figure 5.12 we illustrate the application domain, a 2-dimensional nearest-neighbor with $D_1 = 10$ and $D_2 = 6$, where D_1 and D_2 are the sizes of the X and Y dimensions of the nearest-neighbor grid, respectively. The mapping is described by a tuple of numbers $\{d_1^1, d_2^1\}$, where $d_1^1 | D_1$ (D_1 is a multiple of d_1^1) and $d_2^1 | D_2$ (D_2 is a multiple of d_2^1). The tuple defines the size of the application sub-domain. As the hardware sub-domain in the case of a full-mesh topology is the switch, then $d_1^1 \cdot d_2^1 = p$. The indexing of the nodes in an application sub-domain follows a linear mapping as shown in the figure for the indexing of the sub-domains S_i.

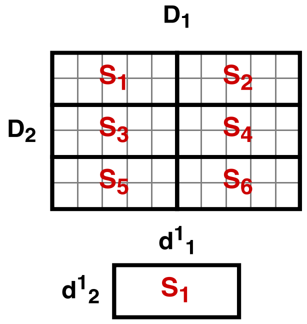


Figure 5.12 – 2D NNE application domains.

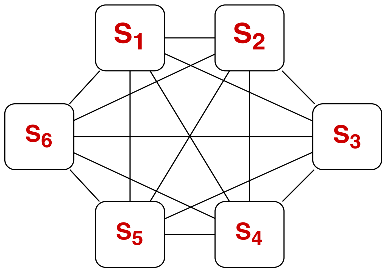


Figure 5.13 – Full-mesh hardware domains.

5.6.2 Full-Mesh Topology

Using the mapping strategy in Figures 5.12 and 5.13, we present bandwidth models for all D_1, D_2, d_1^1, d_2^1 cases, except for the case when $D_1 = d_1^1$ and $D_2 = d_2^1$ which would mean a topology

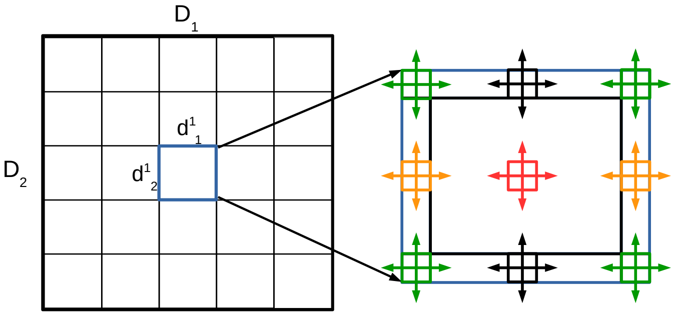


Figure 5.14 – Partition of a 2-dimensional nearest-neighbor application domain.

with only 1 switch, not representative of large-scale systems. We start the analysis with the case when $\boldsymbol{D_1 > 2 \cdot d_1^1}$ **and** $\boldsymbol{D_2 > 2 \cdot d_2^1}$.

We first calculate the effective bandwidth of the $d_1^1 \times d_2^1$ application sub-domain shown in Figure 5.14. All nodes in the sub-domain are connected to the same switch. Each node generates four flows of data to their North/South/East/West neighbors. Depending on where in the sub-domain the nodes are located, they communicate only with nodes within their sub-domain or also with nodes outside of their sub-domain, thus connected to other switches.

The red nodes in the center of the application sub-domain communicate only with nodes connected to the same switch. The green nodes in the corners communicate with two nodes within the same switch and with two nodes connected to two other different switches. The yellow nodes on the East/West edges communicate with three nodes within the same switch and with one node connected to another switch. Finally, the black nodes on the North/South edges of the sub-domain communicate similarly to the yellow nodes on the East/West edges. We will analyze the flows of each of these node types separately and then apply our bandwidth methodology in Section 5.3 to determine $B_{\text{node}}^{\text{eff}}$.

The four flows generated by the red nodes in the center of the sub-domain send messages through two links: source-switch and switch-destination. The effective bandwidth of these flows is calculated as the minimum between the two effective bandwidths achieved on each of the links. Both links are shared by four flows of data, therefore, they are equal to $\frac{b_0}{4}$. The effective bandwidth is thus $B_1 = \frac{b_0}{4}$. The total number of such flows is $n_1 = 4 \cdot (d_1^1 - 2) \cdot (d_2^1 - 2)$.

The four flows generated by the black nodes on the North/South edges of the sub-domain are of two types. (1) Three flows send messages to three nodes within the same switch through two links, source-switch and switch-destination, at $B_2 = \frac{b_0}{4}$ bandwidth each. The total number of such flows is $n_2 = 3 \cdot (2 \cdot d_1^1 - 4)$. (2) The forth flow sends messages to a node connected to another switch through three links: source-switch, switch-switch and switch-destination. The effective bandwidth of the flow is calculated as the minimum across the three effective bandwidths achieved on each of these links. The effective bandwidth on the source-switch and switch-destination links is $\frac{b_0}{4}$. The switch-switch link is shared by d_1^1 flows generated by the d_1^1 nodes on the North/South edge of the sub-domain. Therefore, the effective bandwidth on the switch-switch link is $\frac{b_1}{d_1^1}$. The effective bandwidth of these flows is $B_3 = \min(\frac{b_0}{4}, \frac{b_1}{d_1^1})$. The total number of such flows is $n_3 = 2 \cdot d_1^1 - 4$.

Similarly, we calculate the effective bandwidths of the yellow nodes on the East/West edges of the sub-domain. (1) Three flows send messages to three nodes within the same switch through two links source-switch and switch-destination at $B_4 = \frac{b_0}{4}$ bandwidth. The total number of such flows is $n_4 = 3 \cdot (2 \cdot d_2^1 - 4)$. (2) The forth flow sends messages to a node connected to another switch through three links: source-switch, switch-switch and switch-destination. The effective bandwidth of these flows is $B_5 = \min(\frac{b_0}{4}, \frac{b_1}{d_2^1})$. The total number of such flows is $n_5 = 2 \cdot d_2^1 - 4$.

We also calculate the effective bandwidths of the nodes in the corners of the sub-domain. These nodes generate two types of flows as follows. (1) Two flows send messages to nodes within the same switch through two links source-switch and switch-destination at $B_6 = \frac{b_0}{4}$ bandwidth. The total number of such flows is $n_6 = 2 \cdot 4$. (2) The third flow sends messages to a node connected to another switch through three links: source-switch, switch-switch and switch-destination. The effective bandwidth of this flow is $B_7 = \min(\frac{b_0}{4}, \frac{b_1}{d_1^1})$. The total number of such flows is $n_7 = 1 \cdot 4$. (3) The forth flow sends messages to a node connected to another switch through three links: source-switch, switch-switch and switch-destination. The effective bandwidth of this flow is $B_8 = \min(\frac{b_0}{4}, \frac{b_1}{d_2^1})$. The total number of such flows is $n_8 = 1 \cdot 4$.

All application sub-domains are equivalent, therefore, the same bandwidths B_i and flow counts n_i are valid for all of them. Thus, we calculate the application effective bandwidth as in Equation 5.61, where V is the amount of data per flow.

$$B_{\text{app}}^{\text{eff}} = \frac{\sum_{i=1}^{8} V \cdot n_i}{\max_j(\frac{V}{B_j})} = 4 \cdot D_1 \cdot D_2 \cdot \min_j(B_j) = 4 \cdot D_1 \cdot D_2 \cdot \min\left(\frac{b_0}{4}, \frac{b_1}{d_1^1}, \frac{b_1}{d_2^1}\right) \tag{5.61}$$

We calculate the effective bandwidth per node for the case when $D_1 > 2 \cdot d_1^1$ and $D_2 > 2 \cdot d_2^1$ as:

$$B_{\text{node}}^{\text{eff}} = \frac{B_{\text{app}}^{\text{eff}}}{D_1 \cdot D_2} = 4 \cdot \min\left(\frac{b_0}{4}, \frac{b_1}{d_1^1}, \frac{b_1}{d_2^1}\right). \tag{5.62}$$

For the remaining cases of the combinations of D_1 (equal to d_1^1, equal to $2 \cdot d_1^1$, or greater than $2 \cdot d_1^1$) and D_2 (equal to d_2^1, equal to $2 \cdot d_2^1$, or greater than $2 \cdot d_2^1$), we directly show $B_{\text{node}}^{\text{eff}}$.

If $\boldsymbol{D_1 = 2 \cdot d_1^1}$ **and** $\boldsymbol{D_2 > 2 \cdot d_2^1}$, the West flows of the nodes on the West edges and the East flows of the nodes on the East edges will communicate with their neighbors through the same switch – switch link, thus shared by $2 \cdot d_2^1$ flows. In this case, $B_{\text{node}}^{\text{eff}}$ will be $4 \cdot \min\left(\frac{b_0}{4}, \frac{b_1}{d_1^1}, \frac{b_1}{2 \cdot d_2^1}\right)$.

Similarly, if $\boldsymbol{D_1 > 2 \cdot d_1^1}$ and $\boldsymbol{D_2 = 2 \cdot d_2^1}$, then $B_{\text{node}}^{\text{eff}}$ will be $4 \cdot \min\left(\frac{b_0}{4}, \frac{b_1}{2 \cdot d_1^1}, \frac{b_1}{d_2^1}\right)$. For the case when $\boldsymbol{D_1 = 2 \cdot d_1^1}$ **and** $\boldsymbol{D_2 = 2 \cdot d_2^1}$, $B_{\text{node}}^{\text{eff}}$ will simply be $4 \cdot \min\left(\frac{b_0}{4}, \frac{b_1}{2 \cdot d_1^1}, \frac{b_1}{2 \cdot d_2^1}\right)$.

$\mathbf{D_1}$	$\mathbf{D_2}$	$\boldsymbol{B_{\mathrm{node}}^{\mathrm{eff}}}$
$D_1 > 2 \cdot d_1^1$	$D_2 > 2 \cdot d_2^1$	$4 \cdot \min\left(\frac{b_0}{4}, \frac{b_1}{d_1^1}, \frac{b_1}{d_2^1}\right)$
$D_1 = 2 \cdot d_1^1$	$D_2 > 2 \cdot d_2^1$	$4 \cdot \min\left(\frac{b_0}{4}, \frac{b_1}{d_1^1}, \frac{b_1}{2 \cdot d_2^1}\right)$
$D_1 > 2 \cdot d_1^1$	$D_2 = 2 \cdot d_2^1$	$4 \cdot \min\left(\frac{b_0}{4}, \frac{b_1}{2 \cdot d_1^1}, \frac{b_1}{d_2^1}\right)$
$D_1 = 2 \cdot d_1^1$	$D_2 = 2 \cdot d_2^1$	$4 \cdot \min\left(\frac{b_0}{4}, \frac{b_1}{2 \cdot d_1^1}, \frac{b_1}{2 \cdot d_2^1}\right)$
$D_1 = d_1^1$	$D_2 > 2 \cdot d_2^1$	$4 \cdot \min\left(\frac{b_0}{4}, \frac{b_1}{d_1^1}\right)$
$D_1 > 2 \cdot d_1^1$	$D_2 = d_2^1$	$4 \cdot \min\left(\frac{b_0}{4}, \frac{b_1}{d_2^1}\right)$
$D_1 = 2 \cdot d_1^1$	$D_2 = d_2^1$	$4 \cdot \min\left(\frac{b_0}{4}, \frac{b_1}{2 \cdot d_2^1}\right)$

Table 5.5 – $B_{\mathrm{node}}^{\mathrm{eff}}$ for the full-mesh topology.

Moreover, if $\boldsymbol{D_1 = d_1^1}$ **and** $\boldsymbol{D_2 > 2 \cdot d_2^1}$, then there will be no flow contention on the switch – switch links due to East-West nearest-neighbor communication and $B_{\mathrm{node}}^{\mathrm{eff}} = 4 \cdot \min\left(\frac{b_0}{4}, \frac{b_1}{d_1^1}\right)$. Similarly, if $\boldsymbol{D_1 > 2 \cdot d_1^1}$ **and** $\boldsymbol{D_2 = d_2^1}$, then there will be no flow contention on the switch – switch links due to North-South nearest-neighbor communication and $B_{\mathrm{node}}^{\mathrm{eff}} = 4 \cdot \min\left(\frac{b_0}{4}, \frac{b_1}{d_2^1}\right)$.

The cases of $D_1 = d_1^1$ and $D_2 = 2 \cdot d_2^1$, and $D_1 = 2 \cdot d_1^1$ and $D_2 = d_2^1$ are not representative of large-scale systems. Nevertheless, for $\boldsymbol{D_1 = d_1^1}$ **and** $\boldsymbol{D_2 = 2 \cdot d_2^1}$, $B_{\mathrm{node}}^{\mathrm{eff}} = 4 \cdot \min\left(\frac{b_0}{4}, \frac{b_1}{2 \cdot d_1^1}\right)$, and for $\boldsymbol{D_1 = 2 \cdot d_1^1}$ **and** $\boldsymbol{D_2 = d_2^1}$, $B_{\mathrm{node}}^{\mathrm{eff}} = 4 \cdot \min\left(\frac{b_0}{4}, \frac{b_1}{2 \cdot d_2^1}\right)$. We summarize the bandwidth models for the different mapping scenarios for the full-mesh topology in Table 5.5.

5.6.3 2-Level Fat-Tree Topology

Similarly to the full-mesh topology, the mapping of application processes to hardware nodes is a tuple $\{d_1^1, d_2^1\}$, where $d_1^1 | D_1$ and $d_2^1 | D_2$. In the case of the 2-level fat-tree, each application sub-domain of size $d_1^1 \times d_2^1$ is mapped to a hardware sub-domain which is an L1 switch and $d_1^1 \cdot d_2^1 = m_1$ and $D_1 \cdot D_2 = m_1 \cdot m_2$. As in the case of full-mesh topology, application sub-domains are mapped to the 2-level fat-tree topology linearly.

We present bandwidth models for all D_1, D_2, d_1^1, d_2^1 cases, except for the case when $D_1 = d_1^1$ and $D_2 = d_2^1$ which would mean a topology with only 1 switch, not representative of large-scale systems. We start the analysis with the case when $\boldsymbol{D_1 \geq 2 \cdot d_1^1}$ **and** $\boldsymbol{D_2 \geq 2 \cdot d_2^1}$.

We first calculate the effective bandwidth of the $d_1^1 \times d_2^1$ application sub-domain shown in Figure 5.14. All nodes in the sub-domain are connected to the same L1 switch. Each node generates four flows of data to their North/South/East/West neighbors. Depending on where in the sub-domain the nodes are located, they could communicate only with nodes within their sub-domain or also with nodes outside of their sub-domain, thus connected to other L1 switches.

The red nodes in the center of the application sub-domain communicate only with nodes connected to the same L1 switch. The four flows generated by these nodes send messages through two links: source-switch$_{\text{L1}}$ and switch$_{\text{L1}}$-destination. The effective bandwidth of these flows is calculated as the minimum between the two effective bandwidths achieved on each of the links. Both links are shared by four flows of data, therefore, they are equal to $\frac{b_0}{4}$. The effective flow bandwidth is thus $B_1 = \frac{b_0}{4}$.

The black nodes on the North/South edges communicate with three nodes within the same L1 switch and with one node connected to another L1 switch. The four flows generated by these are of two types. (1) Three flows send messages to three nodes within the same switch through two links, source-switch$_{\text{L1}}$ and switch$_{\text{L1}}$-destination, at $B_2 = \frac{b_0}{4}$ bandwidth. (2) The forth flow sends messages to a node connected to another switch through four links: source-switch$_{\text{L1}}$, switch$_{\text{L1}}$-switch$_{\text{L2}}$, switch$_{\text{L2}}$-switch$_{\text{L1}}$ and switch$_{\text{L1}}$-destination. The effective bandwidth of the flow is calculated as the minimum across the four effective bandwidths achieved on each of these links.

The effective flow bandwidth on the source-switch$_{\text{L1}}$ and switch$_{\text{L1}}$-destination links is $\frac{b_0}{4}$. The w_1 switch$_{\text{L1}}$-switch$_{\text{L2}}$ links are shared by $2 \cdot d_1^1$ flows generated by the $2 \cdot d_1^1$ nodes on the North/South edges, but also by $2 \cdot d_2^1$ flows generated by the $2 \cdot d_2^1$ nodes on the West/East edges of the application sub-domain. Therefore, the effective flow bandwidth on a switch$_{\text{L1}}$-switch$_{\text{L2}}$ link is $\frac{b_1 \cdot w_1}{2 \cdot (d_1^1 + d_2^1)}$. The nodes in a hardware sub-domain are also the destinations of $2 \cdot (d_1^1 + d_1^2)$ flows that share w_1 switch$_{\text{L2}}$-switch$_{\text{L1}}$ links. Thus, the effective flow bandwidth obtained on a switch$_{\text{L2}}$-switch$_{\text{L1}}$ link is $\frac{b_1 \cdot w_1}{2 \cdot (d_1^1 + d_2^1)}$.

Consequently, the effective bandwidth of a flow generated by the black nodes on the North/-South edges of the application sub-domain is $B_3 = \min(\frac{b_0}{4}, \frac{b_1 \cdot w_1}{2 \cdot (d_1^1 + d_2^1)})$. Similarly we calculate the effective flow bandwidths of the remaining nodes (yellow and green in the application sub-domain in Figure 5.14) and we obtain $\frac{b_0}{4}$ for the flows within the hardware sub-domain and $\min(\frac{b_0}{4}, \frac{b_1 \cdot w_1}{2 \cdot (d_1^1 + d_2^1)})$ for the flows that exit their hardware sub-domain.

All application sub-domains are equivalent, thus the same flow bandwidths are valid for all of them. We can calculate the effective bandwidth per application as:

$$B_{\text{app}}^{\text{eff}} = 4 \cdot D_1 \cdot D_2 \cdot \min\left(\frac{b_0}{4}, \frac{b_1 \cdot w_1}{2 \cdot (d_1^1 + d_2^1)}\right). \tag{5.63}$$

We calculate $B_{\text{node}}^{\text{eff}}$ for the case when $D_1 \geq 2 \cdot d_1^1$ and $D_2 \geq 2 \cdot d_2^1$ as follows:

$$B_{\text{node}}^{\text{eff}} = \frac{B_{\text{app}}^{\text{eff}}}{D_1 \cdot D_2} = 4 \cdot \min\left(\frac{b_0}{4}, \frac{b_1 \cdot w_1}{2 \cdot (d_1^1 + d_2^1)}\right). \tag{5.64}$$

For the remaining cases of the combinations of D_1 (equal to d_1^1, or greater than or equal to $2 \cdot d_1^1$) and D_2 (equal to d_2^1, or greater than or equal to $2 \cdot d_2^1$), we directly show $B_{\text{node}}^{\text{eff}}$.

$\mathbf{D}_1$	$\mathbf{D}_2$	$B_{\mathbf{node}}^{\mathbf{eff}}$
$D_1 \geq 2 \cdot d_1^1$	$D_2 \geq 2 \cdot d_2^1$	$4 \cdot \min\left(\frac{b_0}{4}, \frac{b_1 \cdot w_1}{2 \cdot (d_1^1 + d_2^1)}\right)$
$D_1 = d_1^1$	$D_2 \geq 2 \cdot d_2^1$	$4 \cdot \min(\frac{b_0}{4}, \frac{b_1 \cdot w_1}{2 \cdot d_1^1})$
$D_1 \geq 2 \cdot d_1^1$	$D_2 = d_2^1$	$4 \cdot \min(\frac{b_0}{4}, \frac{b_1 \cdot w_1}{2 \cdot d_2^1})$
$D_1 = d_1^1$	$D_2 = d_2^1$	$4 \cdot \frac{b_0}{4}$

Table 5.6 – $B_{\text{node}}^{\text{eff}}$ for the 2-level fat-tree topology.

If $\boldsymbol{D_1 = d_1^1}$ **and** $\boldsymbol{D_2 \geq 2 \cdot d_2^1}$, all the nodes on the East/West edges of the application sub-domain communicate within the switch. Only the North/South nodes generate one flow each to their North/South neighbors that exits its L1 switch and $B_{\text{node}}^{\text{eff}} = 4 \cdot \min(\frac{b_0}{4}, \frac{b_1 \cdot w_1}{2 \cdot d_1^1})$.

Otherwise, if $\boldsymbol{D_1 \geq 2 \cdot d_1^1}$ **and** $\boldsymbol{D_2 = d_2^1}$, all the nodes on the North/South edges of the application sub-domain communicate within the same L1 switch. Only the East/West nodes generate one flow each to their East/West neighbors that exits its L1 switch and $B_{\text{node}}^{\text{eff}} = 4 \cdot \min(\frac{b_0}{4}, \frac{b_1 \cdot w_1}{2 \cdot d_2^1})$.

Finally, if $\boldsymbol{D_1 = d_1^1}$ **and** $\boldsymbol{D_2 = d_2^1}$, none of the flows exits the hardware sub-domain. Thus, $B_{\text{node}}^{\text{eff}} = 4 \cdot \frac{b_0}{4}$. We summarize the bandwidth models for the different mapping scenarios for the 2-level fat-tree topology in Table 5.6.

5.6.4 3-Level Fat-Tree Topology

For the 3-level fat-tree topology, the mapping of application processes to hardware nodes is a quadruple $\{d_1^1, d_2^1, d_1^2, d_2^2\}$, where $d_1^2 | d_1^1 | D_1$ and $d_2^2 | d_2^1 | D_2$ as shown in Figures 5.15 and 5.16 (where $d|D$ means that D is a multiple of d). In Figure 5.15, we show the application domain $D_1 \times D_2$ divided in red sub-domains each of size $d_1^1 \times d_2^1$. Each red sub-domain is further divided in blue sub-sub-domains each of size $d_1^2 \times d_2^2$. For the remainder of this section, we will refer to an application sub-domain as to a 1st-level application sub-domain and to an application sub-sub-domain as to a 2nd-level application sub-domain (the same for the hardware sub-domain and sub-sub-domain, to which we will refer as 1st-level and 2nd-level hardware sub-domain, respectively). In Figure 5.16, we show the hardware domain and the mapping of the red 1st-level application sub-domain to the 1st-level hardware sub-domain and the blue 2nd-level application sub-domain to the 2nd-level hardware sub-domain.

The red 1st-level application sub-domain of size $d_1^1 \times d_2^1$ is mapped to a 1st-level hardware sub-domain which is a $\text{switch}_{\text{L2}}$-rooted sub-tree (L2 is the second level of switching in the 3-level fat-tree topology). The blue 2nd-level application sub-domain of size $d_1^2 \times d_2^2$ is mapped to a 2nd-level hardware sub-domain which is a $\text{switch}_{\text{L1}}$ of the 1st-level hardware sub-domain (L1 is the first level of switching at the bottom of the 3-level fat-tree topology). Moreover, $D_1 \cdot D_2 = m_1 \cdot m_2 \cdot m_3$, $d_1^1 \cdot d_2^1 = m_1 \cdot m_2$, and $d_1^2 \cdot d_2^2 = m_1$. The 1st-level application sub-domains are mapped to the 1st-level hardware sub-domains linearly as well as the 2nd-level application sub-domains within a 1st-level application sub-domain.

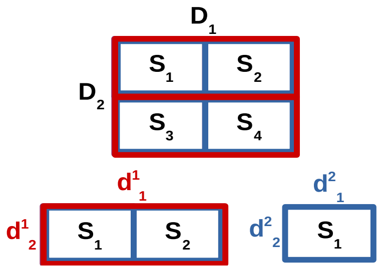


Figure 5.15 – The application domain of a 2-dimensional nearest neighbor pattern. The MPI processes are arranged in a 2-dimensional grid of size $D_1 \times D_2$. The grid is split into sub-grids (1st-level application sub-domains) of size $d^1_1 \times d^1_2$. Each such sub-grid is further divided into sub-grids (2nd-level application sub-domains) of size $d^2_1 \times d^2_2$.

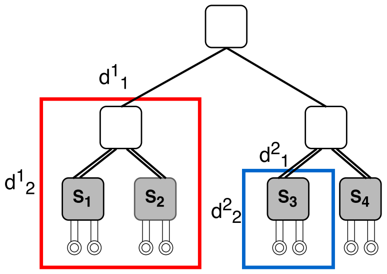


Figure 5.16 – The hardware domain of a 3-level fat-tree. The processes of a 1st-level application sub-domain are linearly mapped to the nodes of one of the m_3 switch$_{\text{L2}}$-rooted sub-trees (1st-level hardware sub-domain). The processes of the 2nd-level application sub-domain are linearly mapped to the nodes of one of the m_2 switch$_{\text{L1}}$ (2nd-level hardware sub-domain) of the switch$_{\text{L2}}$-rooted sub-tree.

For the remainder of this section, we will make the next notations: $D_1 = a \cdot d^1_1$, $D_2 = b \cdot d^1_2$, $d^1_1 = x \cdot d^2_1$, and $d^2_1 = y \cdot d^2_2$. We analyze in detail the case when $\boldsymbol{a \geq 2, b \geq 2, x \geq 2}$ **and** $\boldsymbol{y \geq 2}$. For the remaining cases, we will directly show the final $B^{\text{eff}}_{\text{node}}$ models at the end of this section. There will be two cases that we do not cover: when $x = 1$ and $y = 1$, in which case $m_2 = 1$ and $a = 1$ and $b = 1$, in which case $m_3 = 1$, both cases not being realistic.

Let's start with the flow analysis of the nodes in the 2nd-level application sub-domain of size $d^2_1 \times d^2_2$. All nodes in this sub-domain are connected to the same switch$_{\text{L1}}$. Each node generates four flows of data to their North/South/East/West neighbors. Depending on where in the 2nd-level application sub-domain the nodes are located and on the placement of the 2nd-level application sub-domain within the 1st-level application sub-domain, the nodes can communicate in three ways: (1) only with nodes within the same 2nd-level hardware sub-domain, (2) with nodes outside of their 2nd-level hardware sub-domain, but still within the same 1st-level hardware sub-domain, or (3) with nodes outside their 1st-level hardware sub-domain.

For the nodes communicating with nodes within the same 2nd-level hardware sub-domain, they will send flows at an effective bandwidth of $\frac{b_0}{4}$. Indeed, such flows will traverse the network through two links, source-switch$_{\text{L1}}$ and switch$_{\text{L1}}$-destination, both links being shared by the four flows North/South/West/East sent/received by each node in the network. The number of flows of this type generated within a 2nd-level application sub-domain is $4 \cdot (d^2_1 - 2) \cdot (d^2_2 - 2)$.

All the remaining flows of a 2nd-level application sub-domain, in number of $2 \cdot (d^2_1 + d^2_2)$ will exit their switch$_{\text{L1}}$. Indeed, $2 \cdot d^2_2$ flows will carry the East/West traffic from the nodes on the East/West edges of the 2nd-level application sub-domain and $2 \cdot d_1^2$ flows will carry the North/South traffic from the nodes on the North/South edges of the 2nd-level application sub-domain. All these $2 \cdot (d_1^2 + d_2^2)$ flows will share the w_1 switch$_{\text{L1}}$-switch$_{\text{L2}}$ links. The effective flow bandwidth obtained on such links is thus $\frac{b_1 \cdot w_1}{2 \cdot (d_1^2 + d_2^2)}$. The same effective flow bandwidth will be obtained on the downward switch$_{\text{L2}}$-switch$_{\text{L1}}$ links. Indeed, the nodes in a 2nd-level application sub-domain located on the North/South/East/West edges of the sub-domain will be destinations for exactly $2 \cdot (d_1^2 + d_2^2)$ flows originating in the neighboring 2nd-level application sub-domains. To conclude, the flows that exit their 2nd-level hardware sub-domain, but remain within their 1st-level hardware sub-domain, will have an effective bandwidth of $\min(\frac{b_0}{4}, \frac{b_1 \cdot w_1}{2 \cdot (d_1^2 + d_2^2)})$.

We have so far analyzed the flow contention for links within a 1st-level application sub-domain (source-switch$_{\text{L1}}$, switch$_{\text{L1}}$-switch$_{\text{L2}}$, switch$_{\text{L2}}$- switch$_{\text{L1}}$ and switch$_{\text{L1}}$-destination). To quantify the flow contention on the switch$_{\text{L2}}$-$_{\text{L3}}$ and switch$_{\text{L3}}$-switch$_{\text{L2}}$ links, we analyze the flows of the 1st-level application sub-domain.

A total of $4 \cdot (d_1^1 - 2) \cdot (d_2^1 - 2)$ flows in a 1st-level application sub-domain will communicate within the 1st-level hardware sub-domain. The remaining $2 \cdot (d_1^1 + d_2^1)$ flows will exit their 1st-level hardware sub-domain and traverse the network through the switch$_{\text{L2}}$-switch$_{\text{L3}}$ and switch$_{\text{L3}}$-switch$_{\text{L2}}$ links. Thus, the $w_1 \cdot w_2$ switch$_{\text{L2}}$-switch$_{\text{L3}}$ links will be shared by $2 \cdot (d_1^1 + d_2^1)$ flows and the effective flow bandwidth achieved on these links will be $\frac{b_2 \cdot w_1 \cdot w_2}{2 \cdot (d_1^1 + d_2^1)}$. The same bandwidth will be obtained on the switch$_{\text{L3}}$-switch$_{\text{L2}}$ links. Indeed, the nodes on the four edges of a 1st-level application sub-domain will be the destinations of exactly $2 \cdot (d_1^1 + d_2^1)$ flows originating in the neighboring 1st-level application sub-domains. To conclude, the flows that exit their 1st-level hardware sub-domain will have an effective bandwidth of $\min(\frac{b_0}{4}, \frac{b_1 \cdot w_1}{2 \cdot (d_1^2 + d_2^2)}, \frac{b_2 \cdot w_1 \cdot w_2}{2 \cdot (d_1^1 + d_2^1)})$, as they will experience all the link bottlenecks that occur in the network.

All 1st-level and 2nd-level application sub-domains are equivalent, thus the same flow bandwidths are valid for all of them. We can calculate the effective bandwidth per application as:

$$B_{\text{app}}^{\text{eff}} = 4 \cdot D_1 \cdot D_2 \cdot \min\left(\frac{b_0}{4}, \frac{b_1 \cdot w_1}{2 \cdot (d_1^2 + d_2^2)}, \frac{b_2 \cdot w_1 \cdot w_2}{2 \cdot (d_1^1 + d_2^1)}\right). \tag{5.65}$$

We calculate $B_{\text{node}}^{\text{eff}}$ for the case when $\mathbf{a} \geq 2$, $\mathbf{b} \geq 2$, $\mathbf{x} \geq 2$ and $\mathbf{y} \geq 2$ as follows:

$$B_{\text{node}}^{\text{eff}} = \frac{B_{\text{app}}^{\text{eff}}}{D_1 \cdot D_2} = 4 \cdot \min\left(\frac{b_0}{4}, \frac{b_1 \cdot w_1}{2 \cdot (d_1^2 + d_2^2)}, \frac{b_2 \cdot w_1 \cdot w_2}{2 \cdot (d_1^1 + d_2^1)}\right). \tag{5.66}$$

For the remaining cases of the combinations of a, b, x and y (each equal to 1, or greater than or equal to 2), we directly show the final $B_{\text{node}}^{\text{eff}}$ models.

If $\boldsymbol{a \geq 2}$**,** $\boldsymbol{b \geq 2}$**,** $\boldsymbol{x = 1}$ **and** $\boldsymbol{y \geq 2}$, the effective bandwidth per node is the same as the one in Equation 5.66. Indeed although on the East/West direction there are no intra-1st-level appli-

a	**b**	**x**	**y**	$\mathbf{B^{eff}_{node}}$
$a \geq 2$	$b \geq 2$	$x \geq 1$	$y \geq 1$	$4 \cdot \min\left(\frac{b_0}{4}, \frac{b_1 \cdot w_1}{2 \cdot (d^2_1 + d^2_2)}, \frac{b_2 \cdot w_1 \cdot w_2}{2 \cdot (d^1_1 + d^1_2)}\right)$
$a = 1$	$b \geq 2$	$x \geq 2$	$y \geq 1$	$4 \cdot \min\left(\frac{b_0}{4}, \frac{b_1 \cdot w_1}{2 \cdot (d^2_1 + d^2_2)}, \frac{b_2 \cdot w_1 \cdot w_2}{2 \cdot d^1_1}\right)$
$a = 1$	$b \geq 2$	$x = 1$	$y \geq 2$	$4 \cdot \min\left(\frac{b_0}{4}, \frac{b_1 \cdot w_1}{2 \cdot d^2_1}, \frac{b_2 \cdot w_1 \cdot w_2}{2 \cdot d^1_1}\right)$
$a \geq 2$	$b = 1$	$x \geq 1$	$y \geq 2$	$4 \cdot \min\left(\frac{b_0}{4}, \frac{b_1 \cdot w_1}{2 \cdot (d^2_1 + d^2_2)}, \frac{b_2 \cdot w_1 \cdot w_2}{2 \cdot d^1_2}\right)$
$a \geq 2$	$b = 1$	$x \geq 2$	$y = 1$	$4 \cdot \min\left(\frac{b_0}{4}, \frac{b_1 \cdot w_1}{2 \cdot d^2_2}, \frac{b_2 \cdot w_1 \cdot w_2}{2 \cdot d^1_2}\right)$

Table 5.7 – $B^{\mathrm{eff}}_{\mathrm{node}}$ for the 3-level fat-tree topology.

cation sub-domain 2nd-level application sub-domain neighbors, there are still neighboring 1st-level application sub-domains, thus transfers still occur between them, transfers that traverse the network through all types of switch-switch links. The same effective bandwidth model applies for the case when **$a \geq 2$, $b \geq 2$, $x \geq 2$** and **$y = 1$**. Indeed, although on the North/-South direction there are no intra-1st-level application sub-domain 2nd-level application sub-domain neighbors, there are still neighboring 1st-level application sub-domains, thus transfers still occur between them.

Otherwise, if **$a = 1$, $b \geq 2$, $x \geq 2$ and $y \geq 2$**, the East/West flows are contained in the same 1st-level application sub-domain, thus only $2 \cdot d^1_1$ flows will exit the 1st-level application sub-domain. In this case, $B^{\mathrm{eff}}_{\mathrm{node}} = 4 \cdot \min\left(\frac{b_0}{4}, \frac{b_1 \cdot w_1}{2 \cdot (d^2_1 + d^2_2)}, \frac{b_2 \cdot w_1 \cdot w_2}{2 \cdot d^1_1}\right)$. Else, if **$a = 1$, $b \geq 2$, $x = 1$ and $y \geq 2$**, then in addition to the previous comment, also within a 2nd-level application sub-domain, only $2 \cdot d^2_1$ flows will exist the 2nd-level hardware sub-domain. Thus, $B^{\mathrm{eff}}_{\mathrm{node}} = 4 \cdot \min\left(\frac{b_0}{4}, \frac{b_1 \cdot w_1}{2 \cdot d^2_1}, \frac{b_2 \cdot w_1 \cdot w_2}{2 \cdot d^1_1}\right)$. Finally, if **$a = 1$, $b \geq 2$, $x \geq 2$ and $y = 1$**, although on the North/-South direction there are no intra-1st-level application sub-domain 2nd-level application sub-domain neighbors, there are still neighboring 1st-level application sub-domains, thus the North/South transfers still traverse the network through all the types of switch-switch links. Therefore, $B^{\mathrm{eff}}_{\mathrm{node}} = 4 \cdot \min\left(\frac{b_0}{4}, \frac{b_1 \cdot w_1}{2 \cdot (d^2_1 + d^2_2)}, \frac{b_2 \cdot w_1 \cdot w_2}{2 \cdot d^1_1}\right)$.

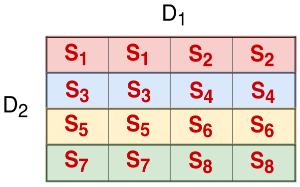
The last cases are for **$a \geq 2$ and $b = 1$**. If **$x \geq 2$ and $y \geq 2$** or **$x = 1$ and $y \geq 2$**, there will be only $2 \cdot d^1_2$ flows exiting the 1st-level application sub-domain. Thus, $B^{\mathrm{eff}}_{\mathrm{node}} = 4 \cdot \min\left(\frac{b_0}{4}, \frac{b_1 \cdot w_1}{2 \cdot (d^2_1 + d^2_2)}, \frac{b_2 \cdot w_1 \cdot w_2}{2 \cdot d^1_2}\right)$. However, when **$x \geq 2$ and $y = 1$**, in addition to the previous comment, there will be only $2 \cdot d^2_2$ flows exiting the 2nd-level application sub-domain. Thus, $B^{\mathrm{eff}}_{\mathrm{node}} = 4 \cdot \min\left(\frac{b_0}{4}, \frac{b_1 \cdot w_1}{2 \cdot d^2_2}, \frac{b_2 \cdot w_1 \cdot w_2}{2 \cdot d^1_2}\right)$. We summarize the bandwidth models for the different mapping scenarios for the 3-level fat-tree topology in Table 5.7.

5.6.5 2-Dimensional HyperX Topology

For the 2D HyperX topology, we cover a set of linear mappings of application sub-domains to hardware sub-domains. The application sub-domain is represented by a full line (row) in the application domain grid. The hardware sub-domain is represented by all the nodes connected to an integer set of m consecutive switches in the horizontal (X) dimension of the network topology ($m \leq d_1$). We bijectively and linearly map application lines to the hardware sub-domains (sequences of switches in the X dimension of the topology). These mapping cases can be described by a number k, where $D_1 \cdot k = d_1 \cdot p$, with $k \in \mathbb{N}_{>0}$ and $D_1 = m \cdot p$, with $m = \frac{d_1}{k} \in \mathbb{N}_{>0}$. As we fully populate nodes in the hardware domain with the processes of the application domain, $D_1 \cdot D_2 = d_1 \cdot d_2 \cdot p$.

We show an example of mapping application sub-domains to hardware sub-domains in Figures 5.17 and 5.18. Figure 5.17 shows a 2D NNE application domain with D_1=4 and D_2=4. Figure 5.18 shows the hardware domain which is a 2D HyperX topology with p=2, d_1=4 and d_2=2. In this case, an application sub-domain is one line of 4 processes in Figure 5.17. A hardware sub-domain is represented by all the nodes connected to 2 consecutive switches in the horizontal (X) dimension of the topology. The figures show a mapping of application sub-domains to hardware sub-domains when m=2 and k=2. m represents the number of switches per hardware sub-domain and k represents the number of hardware sub-domains per X row in the hardware topology. The colors represent the application or hardware sub-domains.

The models herein will cover the cases of $D_2 > 2$ which is common for the 2D NNE pattern.



S_1	S_1	S_2	S_2
S_3	S_3	S_4	S_4
S_5	S_5	S_6	S_6
S_7	S_7	S_8	S_8

Figure 5.17 – 2D NNE application domain.

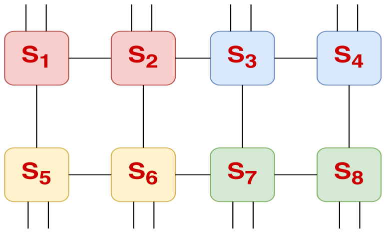


Figure 5.18 – 2D HyperX hardware domain.

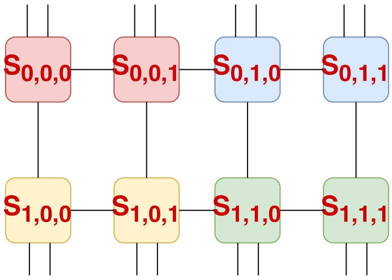


Figure 5.19 – 2D HyperX hardware domain (new switch notation $S_{l,x,c}$).

Throughout this section we will use the following notations. $S_{l,x,c}$ is the c-th switch in the x-th hardware sub-domain of the l-th line (X dimension) of the hardware domain and $N^i_{l,x,c}$ is the i-th node connected to switch $S_{l,x,c}$, with $l \in \{0,1,...(d_2-1)\}$, $x \in \{0,1,...(k-1)\}$, $c \in \{0,1,...(m-1)\}$ and $i \in \{0,1,...(p-1)\}$. Figure 5.19 shows the indexes of the switches in Figure 5.18 when using the $S_{l,x,c}$ notation. For example, S_3 is $S_{0,1,0}$, the first switch in the second hardware sub-domain of the first X row of the hardware domain.

We identify multiple mapping cases depending on the values of k (k equal to 1, equal to 2, or greater than 2) and m (k equal to 1, equal to 2, or greater than 2). In all cases, the nodes send four flows to their corresponding North/South/East/West neighboring nodes. The East/West communications generated by the nodes in a hardware sub-domain H will occur within H and the effective flow bandwidths will be impacted by the m parameter (the number of switches in a hardware sub-domain). However, the North/South communications generated by the nodes in H will occur outside H. We analyze the North/South/East/West flow bandwidths of the nodes of a hardware sub-domain. The analysis is similar for all sub-domains.

For the **East/West** direction communication, for any k value, for all topology lines of switches $l \in \{0,1,...(d_2-1)\}$, for all hardware sub-domains $x \in \{0,1,...(k-1)\}$ in a topology line l and for all switches $c \in \{0,1,...(m-1)\}$ in a hardware sub-domain (l,x), all source nodes $N^i_{l,x,c}$ $\forall i \in \{1,...(p-2)\}$ generate two flows to their East/West neighbors (the destination nodes). Both source and destination nodes are connected to the same switch. The flows thus traverse only two links source-$S_{l,x,c}$ and $S_{l,x,c}$-destination. On both links, the flows achieve an effective bandwidth of $\frac{b_0}{4}$, due to each node sending and receiving exactly four flows of data to and from their North/South/East/West neighbors. For $i \in \{0,(p-1)\}$ and $m=1$, all the flows stay within the switch, thus all East/West flows achieving the same effective bandwidth of $\frac{b_0}{4}$.

Furthermore, for $i \in \{0,(p-1)\}$ and $m \geq 2$, all nodes $N^i_{l,x,c}$ communicate East/West within their hardware sub-domain, but outside their $S_{l,x,c}$. Node $N^0_{l,x,c}$ communicates West via one flow with $N^{p-1}_{l,x,m-1}$ for $c=0$ and node $N^0_{l,x,c}$ communicates West via one flow with $N^{p-1}_{l,x,c-1}$ for $c>0$. Vice-versa, node $N^{p-1}_{l,x,m-1}$ communicates East via one flow with $N^0_{l,x,c}$ for $c=0$ and node $N^{p-1}_{l,x,c-1}$ communicates East also via one flow with $N^0_{l,x,c}$ for $c>0$. These East/West flows traverse three types of links in the hardware sub-domain, source-switch, horizontal switch-switch, switch-destination. The horizontal switch-switch links are shared either only by one flow (if $m>2$) or by two flows (both East and West, if $m=2$). The effective bandwidth of the East/West flows obtained on the horizontal switch-switch links is thus b_1 (if $m>2$) or $\frac{b_1}{2}$ (if $m=2$). In summary, for $i \in \{0,(p-1)\}$ and $m \geq 2$, the effective bandwidth of the East/West flows is calculated as the minimum across the bandwidths obtained on each of the traversing links $\min(\frac{b_0}{4},\frac{b_1}{2})$ if $m=2$, or $\min(\frac{b_0}{4},b_1)$ if $m>2$.

We have so far analyzed the East/West communications for any k and m (each equal to 1, equal to 2, or greater than 2). For the **North/South** direction communication, we start the analysis for $k>2$ and $\forall l \in \{0,1,...(d_2-1)\}$, and $\forall c \in \{0,1,...(m-1)\}$. All source nodes $N^i_{l,x,c}$ $\forall i \in \{0,1,...(p-1)\}$ generate two flows to their North/South neighbors (the destination nodes). For $\forall x \in \{1,...(k-2)\}$, each node $N^i_{l,x,c}$ communicates North via one flow with node $N^i_{l,x-1,c}$ (and vice-versa for the South communication). Moreover, each node $N^i_{l,x,c}$ communicates South via one flow with node $N^i_{l,x+1,c}$ (and vice-versa for the North communication). Thus, a total of p flows share the link between $S_{l,x,c}$ and $S_{l,x-1,c}$ and p flows share the link between $S_{l,x,c}$ and $S_{l,x+1,c}$. Thus, for $\forall x \in \{1,...(k-2)\}$ the horizontal switch-switch links share p flows, each North/South flow achieving on these links an effective bandwidth of $\frac{b_1}{p}$.

m	**k**	$B^{\textbf{eff}}_{\textbf{node}}$
$m = 1$	$k > 2$	$4 \cdot \min(\frac{b_0}{4}, \frac{b_1}{p}, \frac{2 \cdot b_2}{p})$
$m = 1$	$k = 1$	$4 \cdot \min(\frac{b_0}{4}, \frac{b_2}{p})$
$m = 1$	$k = 2$	$4 \cdot \min(\frac{b_0}{4}, \frac{b_1}{2 \cdot p}, \frac{2 \cdot b_2}{p})$
$m = 2$	$k > 2$	$4 \cdot \min(\frac{b_0}{4}, \frac{b_1}{2}, \frac{b_1}{p}, \frac{2 \cdot b_2}{p})$
$m = 2$	$k = 1$	$4 \cdot \min(\frac{b_0}{4}, \frac{b_1}{2}, \frac{b_2}{p})$
$m = 2$	$k = 2$	$4 \cdot \min(\frac{b_0}{4}, \frac{b_1}{2 \cdot p}, \frac{2 \cdot b_2}{p})$
$m > 2$	$k > 2$	$4 \cdot \min(\frac{b_0}{4}, \frac{b_1}{p}, \frac{2 \cdot b_2}{p})$
$m > 2$	$k = 1$	$4 \cdot \min(\frac{b_0}{4}, b_1, \frac{b_2}{p})$
$m > 2$	$k = 2$	$4 \cdot \min(\frac{b_0}{4}, \frac{b_1}{2 \cdot p}, \frac{2 \cdot b_2}{p})$

Table 5.8 – $B^{\text{eff}}_{\text{node}}$ for the 2D HyperX topology.

For $x = 0$, each node connected to $S_{l,0,c}$ communicates North via one flow each with exactly one node connected to $S_{l-1,k-1,c}$ through two equal-length paths: $\frac{p}{2}$ flows through $S_{l,0,c} - S_{l-1,0,c} - S_{l-1,k-1,c}$ and $\frac{p}{2}$ flows through $S_{l,0,c} - S_{l,k-1,c} - S_{l-1,k-1,c}$. However, the $S_{l,0,c} - S_{l,k-1,c}$ link is shared also by $\frac{p}{2}$ flows (North communication) generated by the p nodes of $S_{l+1,0,c}$ to $S_{l,k-1,c}$. The link between $S_{l-1,0,c}$ and $S_{l-1,k-1,c}$ is also shared by $\frac{p}{2}$ flows (North communication) generated by the p nodes of $S_{l-1,0,c}$ to $S_{l-2,k-1,c}$. Thus, the $S_{l,0,c} - S_{l,k-1,c}$ and $S_{l-1,0,c}$-$S_{l-1,k-1,c}$ links are shared by $2 \cdot \frac{p}{2} = p$ flows each. The vertical $S_{l,0,c}$-$S_{l-1,0,c}$ link is shared only by $\frac{p}{2}$ flows (North communication) generated by the p nodes connected to $S_{l,0,c}$, thus each flow sharing this link achieving an effective bandwidth of $\frac{2 \cdot b_2}{p}$. The same analysis is valid for the $S_{l,k-1,c}$-$S_{l-1,k-1,c}$ link. Similar analysis is valid for the case $x = k - 1$. In summary, for $x \in \{0, (k-1)\}$, the North/South flows experience an effective bandwidth of $\min(\frac{b_0}{4}, \frac{b_1}{p}, \frac{2 \cdot b_2}{p})$, as they traverse all the types of links in the network source-switch, horizontal switch-switch, vertical switch-switch and switch-destination.

In the case of $k = 1$, all the nodes $N^i_{l,x,c}$ communicate North with $N^i_{l-1,x,c}$ and South with $N^i_{l+1,x,c}$. Thus, each of the $S_{l,x,c} - S_{l-1,x,c}$ and $S_{l,x,c} - S_{l+1,x,c}$ links is shared by p flows. The North/South flows achieve an effective bandwidth across all the traversed links of $\min(\frac{b_0}{4}, \frac{b_2}{p})$.

Finally, we analyze the case when $k = 2$ and $x \in \{0,1\}$. For $x = 0$, the South communications use the same link $S_{l,0,c} - S_{l,1,c}$ as the North traffic. Indeed, the $S_{l,0,c} - S_{l,1,c}$ link is used by p South flows generated by the nodes $N^i_{l,0,c}$ to the nodes $N^i_{l,1,c}$, by $\frac{p}{2}$ North flows from the nodes $N^i_{l,0,c}$ to the nodes $N^i_{l-1,1,c}$ and by $\frac{p}{2}$ North flows generated by the nodes $N^i_{l+1,0,c}$ to the node $N^i_{l,1,c}$. Thus, the effective bandwidth obtained by both South and North flows on the horizontal switch-switch links is $\frac{b_1}{2 \cdot p}$. The same analysis applies to $x = 1$. The vertical switch-switch links share only $\frac{p}{2}$ flows. Thus, in the case of $k = 2$, the effective bandwidth obtained by the North/South flows is $\min(\frac{b_0}{4}, \frac{b_1}{2 \cdot p}, \frac{2 \cdot b_2}{p})$.

We summarize the cases of all flows and show the models of $B_{\text{node}}^{\text{eff}}$ in Table 5.8.

5.7 Validation Results

In this section, we validate the analytic bandwidth models using the simulation framework presented in [98, 99]. The simulator is based on a discrete event simulation package called Omnest (the commercial version of the OMNet++ package) [9, 8]. The simulator is able to accurately evaluate the performance of custom networks at flit level (the atomic unit of transfer across a communication link) for full-mesh, fat-tree, multi-dimensional tori and 2D HyperX topologies. The framework is highly modular and allows fast customization of the network configuration. The user can set the workload type, the properties of the compute node network adapter, the interconnection network (including the switch hardware characteristics, the flow control and congestion avoidance mechanisms) and the simulation statistics and control. Workloads are represented via stochastic traffic generators whose inter-arrival times, traffic destinations and message sizes can manually be set.

5.7.1 Experimental Setup

For each experiment, we assign a single task per compute node or network adapter. Each node sends at a maximum link bandwidth b_0. For the uniform and shift communication patterns, we assign the tasks to the nodes linearly, meaning that the index of the node is the same with the index of the task. The indexing of the nodes is performed topologically. Starting with zero, all the nodes in a group are indexed consecutively, one group and switch at a time. For the nearest-neighbor pattern, we map the MPI processes following the mapping strategy explained in Section 5.6. We partition the application domain is sub-domains and linearly assign the partitions to the hardware sub-domains. Then, within the application sub-domain, we use linear mapping.

To generate the workload we use the synthetic traffic generator of the network simulator. For the uniform communication pattern, we generate uniform random traffic, where each task selects for each message a destination from all the other tasks with equal probability. For the shift communication pattern, we generate shift traffic, where each MPI process p sends equal-sized messages to another unique process with the index $(p+s) \mod nodes$. For the nearest-neighbor workload, we generate 2-dimensional nearest-neighbor traffic, where each MPI process communicates via equal-sized messages with its North/South/West/East processes (the neighbors as defined in the nearest-neighbor application domain).

For each experiment the metric of interest is the effective bandwidth per node. This is calculated as the total amount of data exchanged divided by the completion time of the communication pattern divided by the number of nodes. For shift and nearest-neighbor, we exchanged a fixed amount of data between every communicating pair (128 KB). For uniform, the nodes communicate uniformly at random for a fixed amount of time (10 milliseconds).

PARAMETER	VALUE
Workload type	Uniform, shift, nearest-neighbor
Message size	512 (bytes)
Network adapter type	Infiniband
Network adapter buffer size	200 (KB)
Network adapter delay	100 (nanoseconds)
Node load	100%
Host-switch bandwidth	7 (GB/second)
Switch flit size	512 (bytes)
Switch type	Infiniband-like
Switch input buffer size	100 (KB per port)
Switch output buffer size	100 (KB per port)
Switch-switch bandwidth	7 (GB/s)

Table 5.9 – Network simulator setup hardware parameters.

For all experiments, we used an input-output-buffered Infiniband switch with 100 KB per buffer as switch architecture. All links are 7 GB/second (FDR) and the flit-size is constant to 512 bytes. Moreover, we used shortest-path routing (dimension-order for tori topologies). Table 5.9 shows a summary of the main network adapter and switch parameters used for our simulations.

5.7.2 Uniform Communication Pattern

We performed experiments for the full-mesh, fat-tree (2-level and 3-level), 2-dimensional HyperX and 3-dimensional torus topologies. The network configurations were chosen so that the number of nodes is approximately 1,000-1,200, with the exception of the tori and HyperX topologies where we covered a larger span of number of nodes from 4 to 3,375.

For the full-mesh topology, we ran simulations with a total number of nodes ∈ {1,026..1,170}, with $p \in \{13..80\}$ and $a \in \{13..80\}$. For the 2-level fat-tree, we ran simulations with a total number of nodes ∈ {1,024..1,224} with $w_0 = 1$, $w_1 \in \{2..39\}$, $m_1 \in \{25..61\}$ and $m_2 \in \{20..42\}$. For the 3-level fat-tree, we performed experiments with a total number of nodes ∈ {1,026..1,200}, with $w_0 = 1$, $w_1 \in \{3..39\}$, $w_2 \in \{2..6\}$, $m_1 \in \{18..61\}$, $m_2 \in \{3..10\}$ and $m_3 = 6$.

For the 3-dimensional torus topology, we ran simulations with a total number of nodes ∈ {8..3,375}, with $p = 1$, $d_1 \in \{2..15\}$, $d_2 \in \{2..64\}$ and $d_3 \in \{2..80\}$. In this design-space we included configurations of 3-dimensional tori topologies with $d_1 = d_2 = d_3 \in \{2..15\}$.

For the 2D HyperX, we performed simulations with a total number of nodes ∈ {16..1,200}, with $p \in \{2..14\}$ and $d_1 = d_2 \in \{4..10\}$. For this particular topology, we only ran HyperX topologies with equal-sized dimensions d_1 and d_2 due to simulator limitations.

Table 5.10 shows the accuracy results of our bandwidth models for the uniform pattern when

Network topology	Mean accuracy	Max accuracy	Min accuracy	Correlation
2L Fat-tree	98.9%	99.3 %	98.7%	0.99
3L Fat-tree	98.8%	99 %	98.6%	0.98
Full-mesh	99.4%	99 %	99.5%	0.99
2D HyperX (same size)	97.9%	99.7%	94.2%	0.99
3D Torus (all)	83.4%	99.5%	38.8%	0.97
3D Torus (same size)	93.1%	99.5%	81.1%	0.99
3D Torus (different sizes)	77.3%	97.3%	38.8%	0.93

Table 5.10 – Accuracy results across network topologies (uniform pattern).

compared with the network simulation results. For each experiment, we calculated the model error as $\frac{|\text{Model}-\text{Simulation}|}{\text{Simulation}}$. The accuracy across a set of experiments (e.g., all experiments corresponding to the same topology) is calculated as 1 minus the average model error across the same set.

The results show that the effective bandwidth models for the uniform communication pattern are very accurate for the fat-tree, full-mesh and 2D HyperX topologies, with average accuracy rates of more than 97%. The 3-dimensional torus topology also exhibits a reasonable accuracy with an average rate of 83% for all torus experiments. Per class of topology, the correlation across the experiments (each experiment corresponding to a network configuration) is also very high of more than 0.97. Moreover, the correlation across the experiments for all topologies is 0.98. This indicates that the proposed effective bandwidth models for the uniform pattern can reliably be used to perform design-space exploration across network configurations.

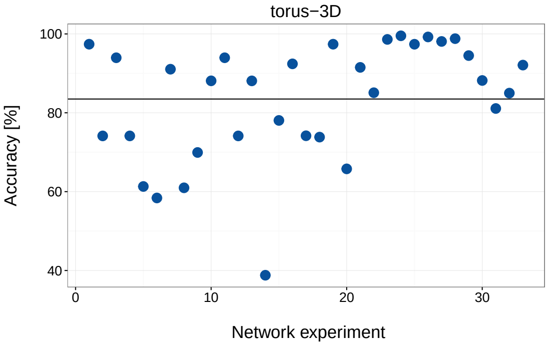


Figure 5.20 – Accuracy variability results for the 3-dimensional torus (uniform pattern).

We further analyzed the 3-dimensional tori results. To understand the variability of the results across different network configurations, we present in Figure 5.20 the accuracy results for all tori experiments. Although the majority of results are well distributed around the mean accuracy, we identify a few configurations with less than 60% accuracy. Through a more detailed investigation of the torus results, we found that tori configurations with equal sized dimensions are more accurately predicted than tori configurations with at least two dimensions of different sizes. Indeed, the average accuracy of tori with equal dimensions is 93%, whereas for the tori with at least two dimensions of different sizes is 77%.

5.7.3 Shift Communication Pattern

We performed experiments for full-mesh and fat-tree (2-level and 3-level) topologies. For the full-mesh topology, we ran simulations with a total number of nodes $\in \{96..1{,}536\}$, with $p \in \{8..64\}$ and $a \in \{12..24\}$. For the 2-level fat-tree, we ran simulations with a total number of nodes $\in \{24..1{,}024\}$ with $w_0 = 1$, $w_1 \in \{2..8\}$, $m_1 \in \{12..64\}$ and $m_2 \in \{2..16\}$. For the 3-level fat-tree, we ran experiments with a total number of nodes $\in \{48..1{,}280\}$, with $w_0 = 1$, $w_1 \in \{2..4\}$, $w_2 \in \{4..8\}$, $m_1 \in \{12..40\}$, $m_2 \in \{2..8\}$ and $m_3 \in \{2..4\}$.

For all topologies, we used shift synthetic traffic with shift-value s. For each network configuration we chose multiple values of s to cover all the cases of s being lower or higher than the sizes of the hardware sub-domains. For example, for full-mesh topology, we ran multiple s values lower or higher than the number of nodes attached to a switch p, for 2-level fat-tree lower or higher than the number of nodes attached to an L1 switch m_1, for 3-level fat-tree lower or higher than the number of nodes attached to an L1 switch m_1, lower or higher than the size of a switch$_{L2}$-rooted sub-tree $m_1 \cdot m_2$.

Table 5.11 shows the accuracy results of the network models for the shift pattern when compared with the network simulation results. For each experiment, we calculated the model error as $\frac{|\text{Model}-\text{Simulation}|}{\text{Simulation}}$. The accuracy across a set of experiments (e.g., all experiments corresponding to the same topology) is calculated as 1 minus the average model error across the same set.

Network topology	Mean accuracy	Max accuracy	Min accuracy	Correlation
2L FAT-TREE	92.8%	99.7%	78%	0.98
3L FAT-TREE	94.1%	99.8%	78%	0.99
FULL-MESH	96.7%	99.8%	82%	0.99

Table 5.11 – Accuracy results across network topologies (shift pattern).

The results show that the effective bandwidth per node for the shift communication pattern are very accurate for the fat-tree and full-mesh topologies, with average accuracy rates of more than 92%. Per class of topology, the correlation across the experiments (same shift-value s, different network configurations) is also very high of more than 0.98. The correlations reported in Table 5.11 correspond to the minimum correlation value obtained across all shift values. This indicates that the proposed effective bandwidth models for the shift pattern can reliably be used to perform design-space exploration across network configurations.

5.7.4 Nearest-Neighbor Communication Pattern

We performed experiments for the full-mesh, 2-level and 3-level fat-tree and 2D HyperX network topologies. For the full-mesh topology, we performed simulations with a total number of nodes 1,024, with $p \in \{8,16,32,64\}$ and $a \in \{16,32,64,128\}$. For the 2-level fat-tree, we ran simulations with a total number of nodes 1,024 with $w_0 = 1$, $w_1 \in \{2,4,8\}$, $m_1 \in \{8,16,32\}$ and $m_2 \in \{32,64,128\}$. For the 3-level fat-tree, we ran experiments with a total number of nodes 1,024, with $w_0 = 1$, $w_1 \in \{2,4\}$, $w_2 = 4$, $m_1 \in \{2,8,32\}$, $m_2 \in \{4,16\}$ and $m_3 \in \{2,8,32,128\}$.

For all the simulations above, we tested all possible mapping strategies with D_1 and D_2 $\in \{2,4,8,16,32,64,128,256,512\}$ and with d_1 and $d_2 \in \{1,2,4,8,16,32,64\}$, where D_1, D_2, d_1 and d_2 satisfy the mapping described in Subsections 5.6.2, 5.6.3 and 5.6.4 for full-mesh, 2-level and 3-level fat-tree topologies, respectively.

For the 2D HyperX topology, we performed simulations with a total number of nodes {100, 200, 300, 400, 500, 1,000}, with $d_1 = d_2 = 10$ and $p \in \{1,2,3,4,5,10\}$. For these simulations, we ran mapping strategies for k and $m \in \{1,2,5,10\}$ following the mapping explained in Subsection 5.6.5.

Table 5.12 shows the accuracy results of the bandwidth models for the 2-dimensional nearest-neighbor communication pattern when compared with the network simulation results. For each experiment, we calculated the model error as $\frac{|\text{Model}-\text{Simulation}|}{\text{Simulation}}$. The accuracy across a set of experiments (e.g., all experiments corresponding to the same topology) is calculated as 1 minus the average model error across the same set.

Network topology	Mean accuracy	Max accuracy	Min accuracy	Correlation
2L FAT-TREE	91.1%	99.8%	80.4%	0.98
3L FAT-TREE	89.7%	99.7%	75%	0.98
FULL-MESH	97.8%	99.8%	95%	0.99
2D HYPERX	83%	91.8%	52%	0.89

Table 5.12 – Accuracy results across network topologies (nearest-neighbor pattern).

The results show that the effective bandwidth per node for the 2-dimensional nearest-neighbor communication pattern are very accurate for the fat-tree and full-mesh topologies, with average accuracy rates of more than 89%. A slightly lower accuracy is obtained for 2D HyperX topologies where we obtain an average of 83% with a minimum of 52%. Per class of topology, the correlation across the experiments (each experiment corresponding to a network configuration and mapping strategy) is also very high of more than 0.98 for fat-tree and full-mesh and 0.89 for 2D HyperX topologies. The results indicate that the proposed effective bandwidth models for the 2-dimensional nearest-neighbor communication pattern can reliably be used to perform design-space exploration across network configurations and mapping strategies.

We further analyze the 2D HyperX results. To understand the variability of the results across different experiments, we present in Figure 5.21 the accuracy results for all 2D HyperX experi-

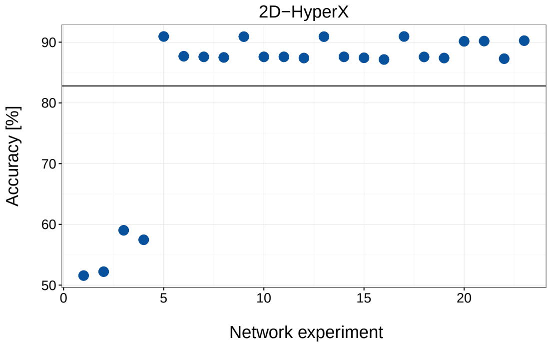


Figure 5.21 – Accuracy variability results for the 2D HyperX topology.

ments. Although the majority of results are well above 80% accuracy, we identify 4 configurations with accuracies between 50 and 60% (which correspond to p=10). We plan as future work to investigate in detail the statistics reported by the 2D HyperX simulations for large values of p in order to understand better where the network contention lies and, if possible, adapt the network models to better capture this network contention.

5.8 Related work

To analyze the network throughput a traditionally used metric is the bisection bandwidth. This metric quantifies the network performance in a communication-pattern-independent manner. It measures the performance as the aggregate bandwidth of the links in a worst-case cut that divides the network in two equal-sized sets of nodes [104]. With our proposed solution we provide analytic bandwidth models that are specific to classes of communication patterns. Our approach provides a more accurate estimate for the effective bandwidth of a node. Indeed, for example, let's assume a full-mesh topology with a switches and p nodes per switch. Assuming that there is no network bottleneck at the end node, the bisection bandwidth is $\frac{b_1 \cdot a^2}{4}$ and the bisection bandwidth per node is $\frac{b_1 \cdot a}{2 \cdot p}$, regardless of the application's communication pattern, routing scheme or MPI rank mapping strategy. In reality, however, if the pattern is uniform, the mapping linear and the routing shortest-path, then, as shown in this chapter, the effective node bandwidth is $\frac{b_1 \cdot a}{p}$. In this case, the bisection bandwidth would significantly underestimate the bandwidth. Another example is if the communication pattern is shift with s a multiple of p. In this case, as also shown in this chapter, the bandwidth estimator is $\frac{b_1}{p}$ which can be very different than the bisection bandwidth.

Simulators are popular approaches to estimate communication performance [129, 98]. Such methods are usually accurate and generic, assuming that the simulators provide implementations for the topologies under study and for the types of traffic of interest. However, they are slow and not scalable to large network sizes or communication execution times. With our methodology, we provide fast means to analyze a large set of hardware network design points. Let's assume that a network system designer or researcher has a large hardware design space to analyze. By using our proposed analytic communication models, the design space can be rapidly analyzed and reduced to a smaller design space with hardware configurations that meet specific requirements. The resulting smaller set of hardware design points can be further analyzed in more detail if necessary using, e.g., simulators or other computationally-intensive more accurate methods.

Another well-known approach to obtain the effective bandwidth is formulating the problem as a flow-optimization problem [121]. Given a network specification and a communication matrix, such a problem will not only provide the bandwidth of the application, but also the routing that ensures it. There is numerous research available on this topic. Kinsy et al. [82] propose an integral max-flow problem by using a mixed integer linear programming (MILP) formulation to find a set of routes for flows that cannot be separated across different paths. The authors state, however, that their optimizations are only feasible from a practical point of view for small network sizes.

Applegate et al. [24] propose a linear programming max-flow formulation for finding close-to-optimal routes that is polynomial in size with respect to the network size. For an Internet Server Provider (backbone) network with 315 routers, they report 31 minutes computation time for their algorithm. Moreover, their work is focusing on estimating network performance when the characteristics of the traffic pattern are not known. The same remark applies to the work presented in [111, 29]. In our work, applications that run on high-performance computing systems usually have a well-defined communication pattern. In our models, we leverage the regularity of such communication patterns in order to create fast means of evaluating their communication performance given a network specification. We assume shortest-path routing and optimal routing when multiple paths are available.

Prisacari et al. [110] propose a highly optimized max-flow integer linear programming (ILP) formulation to efficiently determine optimal routes for arbitrary workloads for fat-tree topologies. Their approach combines ILP with dynamic programming to effectively reduce the time to solution. The authors claim significant speedups over the state of the art. However, even their approach which is specifically optimized for fat-trees topologies, requires hours to weeks for networks of more than 1024 nodes for every single system configuration. These methods are valuable but do not scale, and even if they would become feasible for specific network sizes, larger network sizes would still be unfeasible to analyze. For example, in [110], even though a highly optimized method is proposed, the authors report that when increasing the network size from 512 to 1024, the computation time increases 100-fold. Furthermore, Singla et al. [119] report that their simulator does not scale for all-to-all traffic because the number of commodities in the flow problem increases as the square of the network size.

Prisacari et al. [109] propose a model of the effective bandwidth for dragonfly topologies and the 2-dimensional nearest-neighbor communication pattern. The model includes only link bottlenecks that may occur between dragonfly groups. Our method applies generically to a variety of network topologies and covers all network flows and link bottlenecks. We apply it to multiple combinations of patterns (including 2-dimensional nearest-neighbor) and network topologies in order to provide the scientific community with fast means of design-space exploration across applications and network topologies.

5.9 Conclusions

In this chapter, we proposed a method for estimating the injection bandwidth of a node effectively sustained by a network. The method captures the impact of link contention bottlenecks on the node effective bandwidth. We derived analytic models for the uniform pattern under fat-trees, tori, full-mesh and 2D HyperX topologies, for the shift pattern under full-mesh and fat-tree topologies and for the 2-dimensional nearest-neighbor pattern under fat-trees, full-mesh and 2D HyperX topologies. While for the former two patterns the models apply for linear mapping of the MPI processes to the hardware nodes, for the latter communication pattern the models apply to a larger set of mapping strategies in which application sub-domains are mapped to hardware sub-domains.

The validation results indicate that the proposed effective bandwidth models are not only accurate, but can also reliably be used to perform design-space exploration across network configurations not only across variations of configurations of the same network topology, but also across types of topologies. For the uniform pattern we obtained average accuracy rates of more than 97%, except for tori topologies for which we attained 83%. For the shift pattern we obtained average accuracies of more than 92%. Finally, for the 2-dimensional nearest-neighbor pattern we obtained average accuracies of more than 89% for full-mesh and fat-trees and 83% for 2D HyperX topologies. Moreover, we obtained very high linear correlations of more than 0.89 between the model-based estimates and the simulation-based results for all patterns and network topologies under study.

With these models we provide the community with means of fast design-space exploration across different network topologies. The bandwidth models can also be used with existing tools such as DIMEMAS [32] to allow performance evaluation of parallel application based on detailed compute-communication graphs. As an example, our models could extend DIMEMAS to "analytically simulate" network topologies, by simply replacing the link bandwidth used in the time model of a packet transmission event with the effective bandwidth provided by the analytic models. We will investigate this research path in our future work. As next steps, we also plan to derive similar bandwidth models for other communication patterns such as bit reversal, complement and matrix transpose, typical patterns of HPC applications.

6 Putting it All Together: Full-System Performance Prediction

In the previous chapters we introduced (i) a software profiler that analyzes applications in a platform-agnostic manner, (ii) use cases of how the software properties can enable processor performance evaluation, and (iii) an analytic method for estimating the injection bandwidth of a node in a distributed system, given a communication pattern and a network specification. In this chapter, we combine all the models under a mathematical formulation and evaluate the accuracy of the proposed method with measurements performed on supercomputers. We propose the first analytic methodology for early and fast design-space exploration of large-scale systems, that uses as input hardware- and ISA-agnostic software profiles.

6.1 Introduction

Figure 6.1 shows an overview of our proposed methodology for large-scale system performance modeling. To extract the software model, our methodology uses the platform-independent software analysis (PISA) tool introduced in Chapter 3. PISA measures software properties such as available scalar and vector instruction mix, parallelism, memory access patterns and communication matrix. As the software models are extracted at application run-time, they can only be collected on current systems which are orders of magnitude smaller than exascale. To predict the software models at exascale, our methodology uses the extrapolation tool introduced by Mariani et al. [94], tool that is based on advanced statistical techniques. Once extrapolated, the software model is then loaded into the hardware compute and communication models presented in Chapters 4 and 5. These hardware models analytically capture performance constraints and dependencies of a computing system. The analytic models allow fast exploration of a large design-space of processor and network parameters.

6.2 Full-System Performance Modeling Description

To estimate the execution time of an MPI application, we first cluster the profiles of the MPI processes extracted with PISA in classes of similar compute and communication signatures.

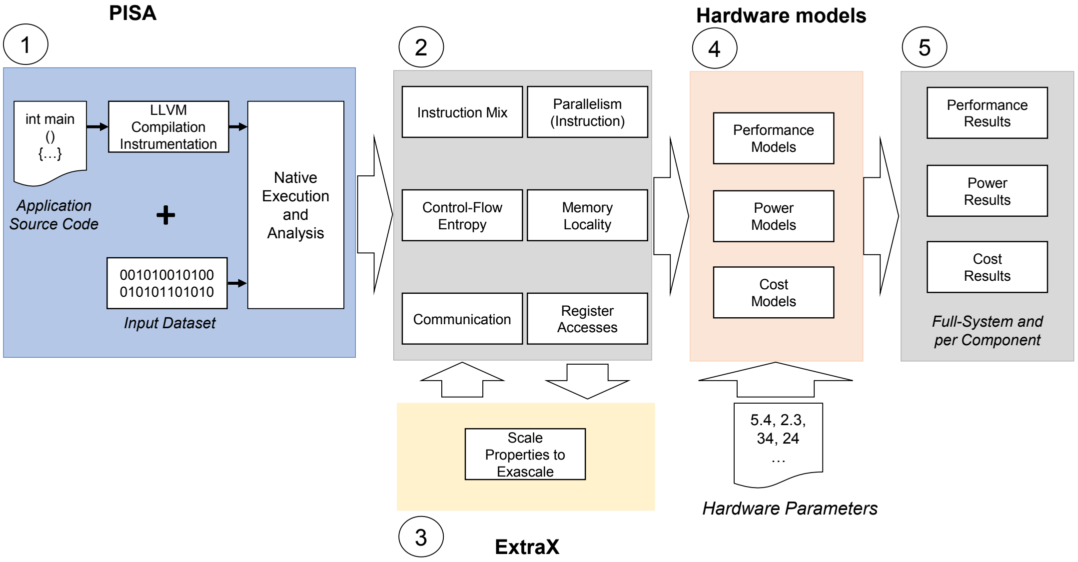


Figure 6.1 – Full-system performance evaluation methodology overview.

The clustering is performed using the method presented by Mariani et al. [95], a tool based on unsupervised machine-learning techniques to consistently classify threads in different program runs. Such a clustering method allows to speed up the performance analysis of large sets of hardware design points. Indeed, instead of calculating performance metrics for each individual thread or process, which in an exascale system can be in the range of hundreds of thousands, our proposed methodology calculates the performance only for the representative processes of the identified classes. This increases the capacity of the methodology at the expense of reduced accuracy.

For each such class of processes C, we analytically estimate the execution time of a representative process of the class as shown in Equation 6.1.

$$\text{Time}^C = \text{Time}^C_{\text{compute}} + \text{Time}^C_{\text{communication}} \tag{6.1}$$

The assumption taken by the model is that there is no overlap between the compute and communication time of the process of class C. To quantify the overlap in time between compute and communication, the application signature should include the temporal sequence of the compute and communication events that occur during the execution of an MPI application. However, the current PISA application model does not include such a compute-communication graph, thus the methodology does not include a model for the overlap time. Although PISA could generate a hardware-independent compute-communication graph, the main reason for not considering the graph is that it would require simulation for evaluating the performance of each different hardware design point. This would significantly limit the design-space exploration capacity of our methodology. Moreover, to generate a communication graph of a problem size and number of processes representative of an exascale system would not be possible for two reasons: the lack of exascale systems and the very large trace size.

The compute time of a representative process of class *C* is calculated based on the PISA compute signature and the architectural parameters of the processor (Equation 6.2) using processor performance models such as those presented in Section 4.2. The PISA signature includes the instruction mix (InstructionMix^C), the overall and per-instruction-type instruction-level parallelism (ILP^C, $\text{ILP}^C_{\text{type}}$) and the temporal memory access patterns (MemoryPattern^C). The architectural parameters refer not only to the architecture of the processor, but also to the architecture of both the cache hierarchy and the external DRAM.

$$\begin{aligned} \text{Time}^C_{\text{compute}} &= \text{f}(\text{SwSignature}^C_{\text{compute}}, \text{HwParameters}_{\text{processor}}) \\ \text{SwSignature}^C_{\text{compute}} &= \{\text{InstructionMix}^C, \text{ILP}^C, \text{ILP}^C_{\text{type}}, \text{MemoryPattern}^C\} \end{aligned} \tag{6.2}$$

The communication time of a representative process of class *C* is calculated based on the PISA communication signature and the architectural parameters of the network topology (Equation 6.3). The PISA communication signature includes three parameters: the regular communication pattern that best approximates the inter-process communication behavior of the application $\text{Pattern}^{\text{App}}_{\text{communication}}$ (e.g., uniform, 2-dimensional nearest-neighbor), the total number of exchanged MPI messages M^C and the average message size M^C_{Size}. The architectural parameters of the network include not only the parameters that describe the structure of the interconnect fabric (e.g., the number of up-links and down-links at each level in a fat-tree, the sizes of the three dimensions of a 3-dimensional torus), but also the communication latencies, the node MPI stack latency, the link latency and the switch latency.

$$\begin{aligned} \text{Time}^C_{\text{communication}} &= \text{g}(\text{SwSignature}^C_{\text{communication}}, \text{HwParameters}_{\text{network}}) \\ \text{SwSignature}^C_{\text{communication}} &= \{\text{Pattern}^{\text{App}}_{\text{communication}}, \text{M}^C, \text{M}^C_{\text{Size}}\} \end{aligned} \tag{6.3}$$

Equation 6.4 shows the actual model for the communication time of a representative process of class *C*, model that considers different sources of communication latency.

$$\text{Time}^C_{\text{communication}} = \text{M}^C \cdot (\text{l}_{\text{node}} + \text{l}_{\text{link}} + \text{s} \cdot \text{l}_{\text{switch}} + \frac{\text{M}^C_{\text{Size}}}{\text{B}^{\text{eff}}_{\text{node}}}) \tag{6.4}$$

The average communication latency per message l_{link} incurred due to link traversal is calculated as the sum of the lengths of all the paths taken by messages exchanged by the MPI processes during the execution of the application divided by the total number of paths. The average communication latency incurred due to switch traversal is calculated as the average number of switches *s* traversed by the MPI messages multiplied by the switch latency l_{switch}. The average number of switches is estimated by subtracting 1 from the average number of links traversed by the MPI messages during the application run.

It is important to note that the average link and switch latency models take into account the communication pattern that best approximates the communication behavior of the application, thus the model is approximately tailored to the application communication pattern. The regularity of the approximated communication pattern allows us to derive closed-form formulas for both average link and switch latencies, without running simulation.

Moreover the communication effective bandwidth $\mathrm{B}_{\mathrm{node}}^{\mathrm{eff}}$ at which the end nodes exchange MPI messages is estimated using the method described in Section 5.3. For each combination of a communication pattern with a network topology the methodology uses a separate analytic model for $\mathrm{B}_{\mathrm{node}}^{\mathrm{eff}}$.

In this section we have so far proposed a set of models to estimate the communication and compute times per class of MPI processes C. We estimate the completion time of the full MPI application across all classes of processes by using Equation 6.5.

$$\mathrm{Time}_{\mathrm{App}} = \max_{C}(\mathrm{Time}_{\mathrm{compute}}^{C} + \mathrm{Time}_{\mathrm{communication}}^{C}) \tag{6.5}$$

We believe that such a model would characterize with reasonable accuracy the completion time of an MPI application with limited waiting time across processes. We consider such an assumption reasonable as programmers of parallel and distributed high-performance computing systems usually aim to balance the workload across the nodes and to reduce the idle times across MPI processes.

6.3 Validation Results

We apply our full-system performance prediction methodology to two MPI benchmarks: Graph 500 [4] and NAS LU [33] for which we also performed a set of time measurements on supercomputers with different network topology configurations. Graph 500 is a representative of graph analytics benchmarks and provides multiple MPI implementations. We analyze the most scalable MPI implementation of the benchmark (MPI-simple). NAS LU is a computational fluid dynamics application of the NAS benchmarks [33]. To estimate the communication time, we use the uniform bandwidth estimators introduced in Section 5.4 for Graph 500 and the 2-dimensional nearest-neighbor bandwidth estimators introduced in Section 5.6 for the NAS LU benchmark. Indeed, as shown in Chapter 2, Graph 500 has a nearly uniform all-to-all [21] pattern and NAS LU has a 2-dimensional nearest-neighbor pattern [95].

To perform real measurements we used three supercomputing systems. One was a Cray XC40 Series Supercomputer (Magnus) provided by the Pawsey Supercomputing Center in Australia [7]. The system was ranked in November 2014 at No. 77 in the world's Green500 most energy-efficient supercomputers list. This supercomputer provided us with access to network configurations of full-mesh and 2D HyperX topologies. The second supercomputer used for our measurements was Orzel / Eagle provided by the Poznan Supercomputing and Network Center (PSNC), a supercomputer with a performance of 1.4 PFlops and a fat-tree network interconnect fabric. This system allowed us to run different network configurations of 2-level and 3-level fat-tree topologies. The third supercomputer was an IBM Blue Gene/Q supercomputer where we had access to a limited set of network configurations of 1-dimensional, 2-dimensional and 3-dimensional tori topologies. We used all three supercomputers to analyze the performance of Graph 500. For NAS LU we used only the first two systems, as the Blue Gene/Q had been decommissioned at the time we started the evaluation of the NAS benchmark.

From a compute hardware perspective, both Magnus and Eagle use x86 Haswell processors, while Blue Gene/Q uses a POWER A2 in-order processor [13]. From a network perspective, we measured the node MPI stack latencies (l_{node}) using the Intel MPI benchmarks [5]. For the link and switch latencies, for the Magnus system we used information provided in [40], for the Eagle system we used information obtained through personal communications with the system administrators and for the Blue Gene/Q system we used information reported in [50]. Regarding link bandwidths, Eagle provides a fat-tree interconnect fabric with 7 GB/s FDR links, Magnus provides a 2D HyperX (and full-mesh) interconnect fabric with 5 GB/s links [7] and the Blue Gene/Q provides a torus fabric with 2 GB/s links [50].

6.3.1 Graph 500 Benchmark

For the *model estimates*, we profiled with PISA the BFS execution of the MPI-simple implementation of the Graph 500 benchmark for graph scales from 16 to 21, for an edge factor of 16 and for different counts of concurrent MPI processes from 4 to 64 (in powers of 2). For larger scales we used ExtraX to extrapolate the software profile to target scale. We used either the PISA or the extrapolated software compute profiles loaded into the processor performance model presented in [76] to estimate the compute time. As processor architectures we used E5-2597 v3 Haswell (for the Magnus and Eagle compute chips) and POWER A2 [67] (for the Blue Gene/Q compute chip).

To analytically estimate the communication time, we need models for the average link latency, average number of switches traversed per MPI message and the effective bandwidth. In Table 1 in the appendix, we show the analytic models for the average link latency per message for the uniform pattern. To determine the average number of links traversed by a message, the analytic models are similar, where the link latencies at the different levels of hierarchy in the network (l_0, l_1, l_2) are replaced with 1. The average number of switches is calculated by subtracting 1 from the average number of links. As bandwidth estimators, we use the effective bandwidth (per node) models derived in Section 5.4 for the uniform communication pattern .

For the *real measurements*, we ran the MPI-simple implementation of Graph 500 for scales from 16 to 25 for an edge factor of 16 with different number of processes from 4 to 64 (in powers of 2), following a linear mapping strategy of assigning one MPI process per compute node. Each of these problem sizes were run on supercomputers with network configurations as shown in Table 2 in the appendix. For each experiment we measured the execution time of the BFS kernel. We compare the execution time measurement with the time analytically estimated using our full-system methodology.

We calculate the accuracy of our models and report the results per system (Eagle with 2-level or 3-level fat-tree topology, Magnus with full-mesh or 2D HyperX topology and Blue Gene/Q with torus topology). For each experiment we calculate the model error as $\frac{|\text{Model}-\text{Measurement}|}{\text{Measurement}}$. The average system error is calculated as the average across all experiments performed on that system. The accuracy is calculated as 1 minus the average system error. Table 6.1 shows the accuracy results obtained when comparing the real measurements with the analytic estimates derived from the combination of PISA software profiles with hardware (processor and network) models (items no. 2 and 4 in Figure 6.1). The results in this section correspond to application problem sizes of scales from 16 to 21, for an edge factor of 16 and for a number of concurrent MPI processes from 4 to 64.

Topology (System)	**Average accuracy**	**Max accuracy**	**Min accuracy**	**Correlation**
2L FAT-TREE (EAGLE)	73.14 %	77 %	59 %	0.99
3L FAT-TREE (EAGLE)	64.66 %	76 %	42 %	0.98
FULL-MESH (MAGNUS)	90.80 %	99.7 %	76 %	0.98
2D HYPERX (MAGNUS)	89.31 %	90 %	72 %	0.98
1D TORUS (BG/Q)	95.41 %	99.6 %	93.2 %	–
2D TORUS (BG/Q)	84.82 %	99.5 %	17 %	0.99
3D TORUS (BG/Q)	67.5 %	98.7 %	21 %	0.95

Table 6.1 – Accuracy results across systems (Graph 500).

The results show that with our proposed methodology we can generally evaluate performance with reasonable accuracy of 90.80% for full-mesh, 89.3% for the 2D HyperX, 73.1% for the 2-level fat-tree, 64.6% for the 3-level fat-tree, 95.4% for the 1-dimensional torus, 84.8% for the 2-dimensional torus and 67.5% for the 3-dimensional torus. The results with the highest variability around the mean accuracy were obtained for systems with torus fabrics. This result may be due to the fact that a torus topology typically uses complex deadlock avoidance mechanisms that can generate additional congestion in the network, that is not covered with our communication bandwidth models.

A more detailed view of the accuracy results is presented in Figure 6.2 where we show the accuracy for all tested hardware configurations. On the X axis we show the network topology specifications in the following format : (fm,p,a) for full-mesh, (ft2L, w_0,w_1,m_1,m_2) for 2-level fat-tree, (ft3L, w_0,w_1,w_2,m_1,m_2,m_3) for 3-level fat-tree, (2dHyperX,p,d_1,d_2) for 2D HyperX, (t1d,p,d_1) for 1D torus, (t2d,p,d_1,d_2) for 2D torus and (t3d,p,d_1,d_2,d_3) for 3D torus topologies.

In order to conclude whether our methodology captures the trends in performance across different hardware configurations, we calculated the correlation factors for the different types of systems. For a given application problem size and system (network) type, we calculate the correlation factor across all the tested hardware configurations of that system. We report in Table 6.1 the minimum correlation value obtained across the application problem sizes. For 2-dimensional and 3-dimensional torus-based systems, we excluded the result obtained for the smallest application problem size (16), where we obtained a negative correlation factor. Nevertheless, for all larger scales, the results indicate that our full-system performance methodology is able for the Graph 500 benchmark to accurately capture the relative trends in performance across different hardware configurations. We do not show the correlation for the 1D torus-based system, as we only had one network configuration available.

We also calculated the correlation factor across all types of systems per application problem size. For the application problem size 16 we obtained a correlation factor of 0.71 and for all the other problem sizes (17, 18, 19, 20, 21) we obtained a correlation factor of more than 0.98. This is again a encouraging result for using our full-system methodology for system performance ranking and early system design-space exploration.

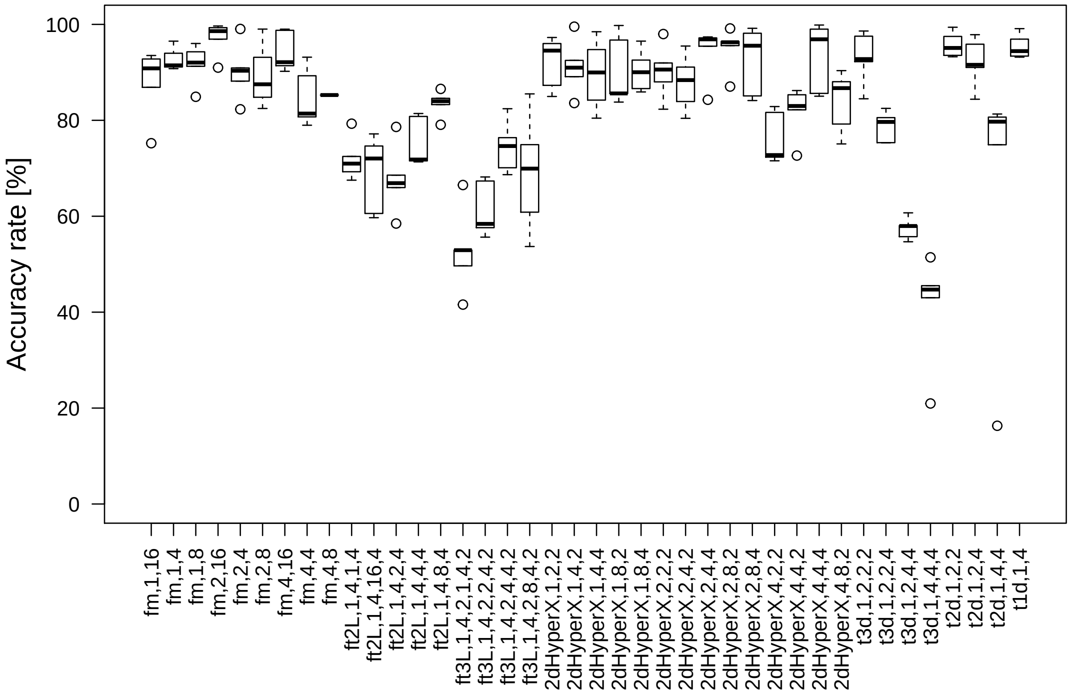


Figure 6.2 – Full-system model validation (PISA and hardware models) – Graph 500.

We also compared the real measurements with extrapolated PISA profiles loaded into hardware models (items no. 1, 3 and 4 in Figure 6.1) and show the results in Table 6.2. The reported results are for application graph scales of 24 and 25, for an edge factor of 16 and a number of processes of 64. We only report the accuracy rates for full-mesh, fat-tree and 2D HyperX systems, as we did not perform measurements for these large graph scales for the tori fabric (which had been decommissioned at the time we ran the large-scale measurements).

The results indicate that the estimates of using extrapolated PISA software profiles with hardware models are reasonably accurate with 78% accuracy for fat-tree topologies and with more than 84% for the full-mesh and 2D HyperX interconnects. A more detailed view of the results per network configuration is shown in Figure 6.3. We also calculated the correlation

Network topology	Problem size 24	Problem size 25
2L FAT-TREE	78.98 %	80.58 %
3L FAT-TREE	78.37 %	76.27 %
FULL-MESH	86.91 %	93.13 %
2D HYPERX	84.3 %	86.3 %

Table 6.2 – Accuracy results across systems (Graph 500).

factors obtained for each of the application problem sizes across hardware configurations. We obtained a correlation factor of 0.97 for scale 24 and 0.92 for scale 25. This is again a good result indicating that our full-system methodology could reliably be used to analyze performance trends when varying the system hardware configurations.

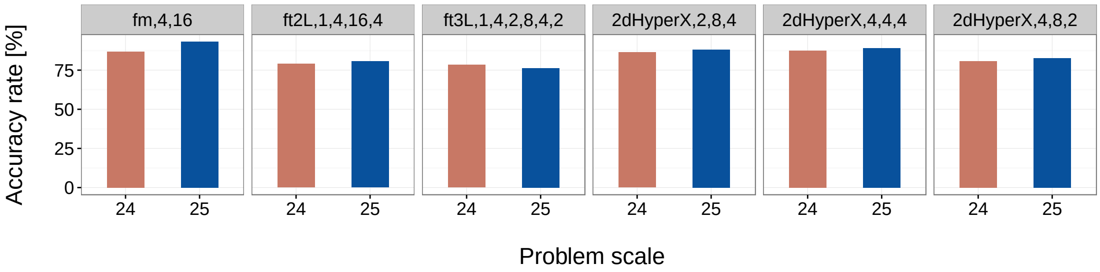


Figure 6.3 – Full-system model validation (PISA, ExtraX and hardware models) – Graph 500.

6.3.2 NAS LU Benchmark

For the *model estimates*, we ran with PISA the NAS LU benchmark for scales 36, 64, 102 and 162 with different counts of processes from 4 to 64 (in powers of 2). For larger scales, we used ExtraX to extrapolate the PISA software profiles obtained at smaller scales to 408 and 800. We used either the PISA or the extrapolated software compute profiles loaded into the processor performance model presented in [76] to estimate the compute time. As processor architectures we used E5-2597 v3 Haswell (for the Magnus and Eagle compute chips).

To analytically estimate the communication time, we need models for the average link latency, average number of switches traversed per MPI message and the effective bandwidth per node. In Tables 3, 4, 5 and 6 in the appendix, we show the analytic models for the average link latency per message for the 2-dimensional nearest-neighbor pattern. To determine the average number of links traversed by a message, the analytic models are similar, where the link latencies at the different levels of hierarchy in the network (l_0, l_1, l_2) are replaced with 1. The average number of switches is calculated by subtracting 1 from the average number of links. As bandwidth estimators, we use the effective bandwidth (per node) models derived in Section 5.6 for the 2-dimensional nearest-neighbor communication pattern .

For the *real measurements*, we ran the NAS LU benchmark for problem sizes 36, 64, 102, 162, 408, 800 with different number of processes from 4 to 64 (in powers of 2), following a linear mapping strategy of assigning one MPI process per compute node. Each of these problem sizes were run on a supercomputer with network configurations as shown in Table 7 in the appendix. For each experiment we measured the execution time of the benchmark. We compare the time measurement with the time analytically estimated using our full-system methodology.

We calculate the accuracy of our models and report the results per system (Eagle with 2-level or 3-level fat-tree topology, Magnus with full-mesh or 2D HyperX topology). For each experiment we calculate the model error as $\frac{|\text{Model}-\text{Measurement}|}{\text{Measurement}}$. The average system error is calculated as the average across all experiments performed on that system. The accuracy is calculated as 1 minus the average system error. Table 6.3 shows the accuracy results obtained when comparing the real measurements with the analytic estimates derived from using PISA software profiles with hardware (processor and network) models (items no. 2 and 4 in Figure 6.1). The results in this section correspond to application problem sizes of 36, 64, 102 and 162 and for a number of concurrent MPI processes from 4 to 64.

Topology (System)	**Average accuracy**	**Max accuracy**	**Min accuracy**	**Correlation**
2L FAT-TREE (EAGLE)	82.4 %	92 %	67 %	0.99
3L FAT-TREE (EAGLE)	72.3 %	87 %	34 %	0.99
FULL-MESH (MAGNUS)	78.7 %	99 %	46 %	0.95
2D HYPERX (MAGNUS)	82 %	99 %	47 %	0.94

Table 6.3 – Accuracy results across systems (NAS LU).

The results show that with our proposed methodology we estimate performance with accuracy of 78.7% for full-mesh, 82% for the 2D HyperX, 82.4% for the 2-level fat-tree, 72.3% for the 3-level fat-tree systems. The results, however, exhibit a larger variability around the mean accuracy than those obtained for Graph 500. A more detailed view of the accuracy rates is presented in Figure 6.4 where we show the accuracy for all tested hardware configurations. On the X axis we show the network topology specifications in the following format : (fm,p,a) for full-mesh, (ft2L, w_0,w_1,m_1,m_2) for 2-level fat-tree, (ft3L, w_0,w_1,w_2,m_1,m_2,m_3) for 3-level fat-tree, (2dHyperX,p,d_1,d_2) for 2D HyperX topologies.

In order to conclude whether our methodology captures the trends in performance across different hardware configurations, we calculated the correlation factors for the different types of systems. For a given application problem size and system (network) type, we calculate the correlation factor across all the tested hardware configurations of that system. We report in Table 6.3 the minimum correlation value obtained across the application problem sizes. We also calculated the correlation factor across all types of systems per application problem size. The lowest correlation factor of 0.92 was obtained for the application problem size 36. The results indicate that our full-system performance methodology is able, similarly to the Graph 500 benchmark, to accurately capture the relative trends in performance across different hardware configurations. This is an encouraging result for using our full-system methodology for system performance ranking and early system design-space exploration.

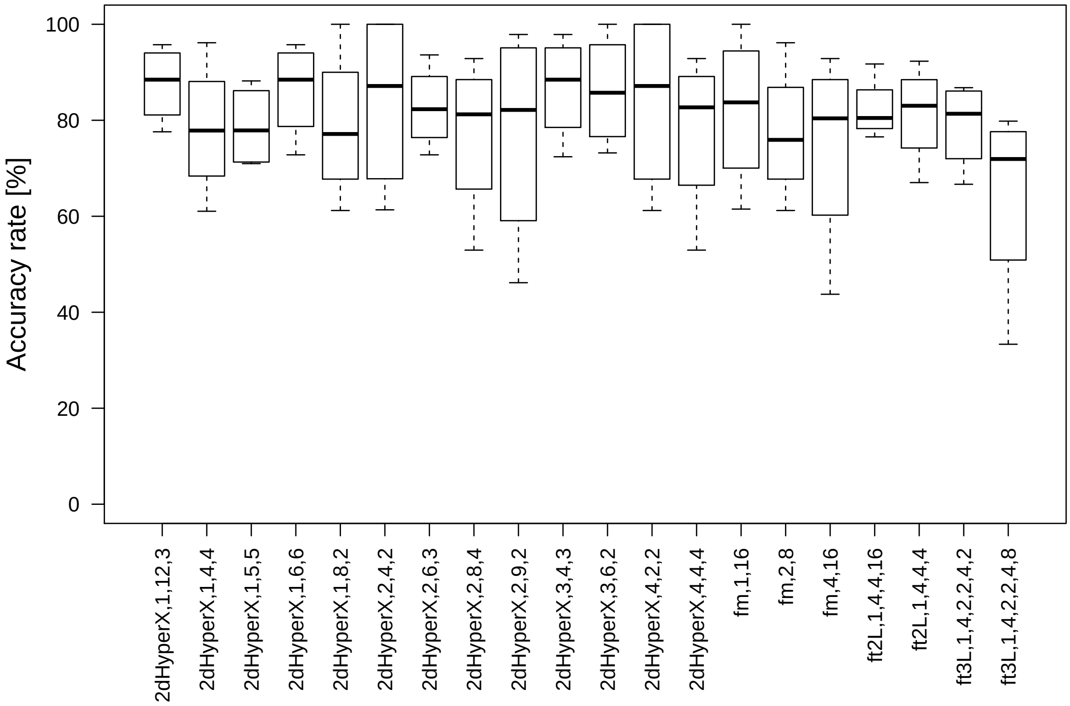


Figure 6.4 – Full-system model validation (PISA and hardware models) – NAS LU.

We also compared the measurements with those obtained from extrapolated PISA profiles loaded into hardware models (items no. 2, 3 and 4 in Figure 6.1) and show the results in Table 6.2 for application problem sizes of 408 and 800 and a number of processes of 64. We obtain 60% accuracy for the problem size of 408 and 92% accuracy for the problem size of 800. A more detailed view of the results per network configuration is shown in Figure 6.5.

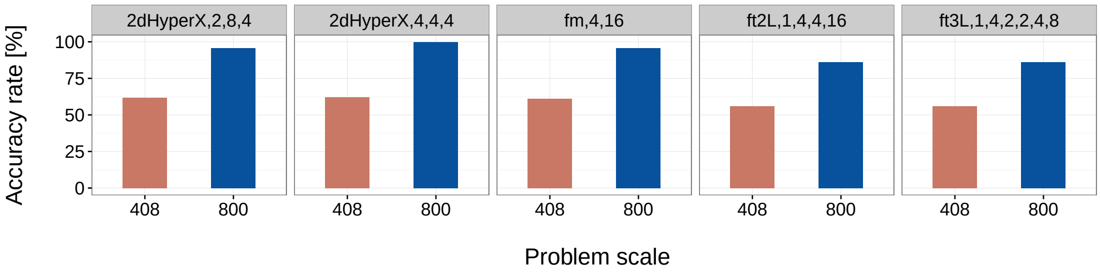


Figure 6.5 – Full-system model validation (PISA, ExtraX and hardware models) – NAS LU.

We also analyzed the correlation factors obtained for each of the problem sizes 408 and 800 across hardware configurations. We find that for NAS LU, when extrapolating the PISA profiles with ExtraX to the larger problem sizes, the system performance ranking across different hardware configurations is no longer preserved. As in the absence of ExtraX, the correlation factors were very high, more investigation is necessary to understand the impact of ExtraX on the quality of the estimates. This represents future work.

6.4 Full-System Power Modeling Description

In this section we introduce an analytic model for estimating the power consumption of a full system (both compute and communication). Although we do not validate this model with power measurements we use it as a first-principle-based power model to perform theoretical trade-off performance-power analysis across hardware design points. We apply this model in the Chapter 7 for the Graph 500 benchmark assuming a large-scale system.

The energy of the system can be calculated as the sum between the communication energy required by the end nodes to exchange messages over the network and the compute energy consumed by all the end nodes (Equation 6.6). The model assumes that an N-process MPI run is executed on N nodes, one MPI process per node.

$$E_{\text{system}} = N \cdot E_{\text{compute}} + E_{\text{communication}} \tag{6.6}$$

Equation 6.6 is valid if all MPI processes have the same compute signature, thus the same processor and memory energy consumption. However, in practice, this assumption may not hold. We can adjust Equation 6.6 to take into account different compute signatures. By employing the process clustering methodology in [95], the MPI processes can be clustered in classes C_n, each class comprising of processes with similar compute and communication signatures. Equation 6.7 shows the system energy model when considering classes of processes, where $|C_n|$ represents the number of processes of class C_n.

$$E_{\text{system}} = \sum_{C_n} |C_n| \cdot E^{C_n}_{\text{compute}} + E_{\text{communication}} \tag{6.7}$$

The compute energy of a process of class C_n is calculated based on the static and dynamic power of the processor and memory evaluated with McPat and CACTI/MeSAP as described in Chapter 4.3. Equation 6.8 shows the model for the compute energy of a process of class C_n. T_{App} is the total execution time of the MPI application as calculated in Equation 6.5 and $T^{|C_n|}_{\text{compute}}$ represents the compute time of a process of class C_n.

$$\begin{aligned} E^{C_n}_{\text{compute}} &= E^{C_n}_{\text{compute-static}} + E^{C_n}_{\text{compute-dynamic}} \\ E^{C_n}_{\text{compute}} &= P^{C_n}_{\text{compute-static}} \cdot T_{\text{App}} + P^{C_n}_{\text{compute-dynamic}} \cdot T^{C_n}_{\text{compute}} \end{aligned} \tag{6.8}$$

We have so far described the compute energy model. We continue in the following paragraphs with the communication energy model. Equation 6.9 shows the energy communication model for a process of class C_n.

$$E_{\text{communication}} = E_{\text{communication-static}} + \sum_{C_n} |C_n| \cdot E^{C_n}_{\text{communication-dynamic}} \tag{6.9}$$

Equation 6.10 shows the static energy of the communication $E_{\text{communication-static}}$. The model accounts for the static energy consumed by all the network switches s during the execution time of the MPI application T_{App}. The total number of switches is calculated based on the network topology configuration parameters.

$$E_{\text{communication-static}} = s \cdot E_{\text{switch-static}} = s \cdot P_{\text{switch-static}} \cdot T_{\text{App}} \tag{6.10}$$

The dynamic energy consumption model quantifies the dynamic energy consumed by the link drivers in the network and the switch logic, energy consumed to transfer the MPI messages from source to destination nodes. Equation 6.11 shows this model, where l, l^{elec}, l^{opt} and s represent the average number of links, the average number of electrical links, the average number of optical links and the average number of switches through which messages are sent, respectively. These model parameter values are calculated based on the network topology configuration and the communication pattern that best approximates the communication behavior of the MPI application.

$$\begin{aligned} E^{C_n}_{\text{communication-dynamic}} &= (l^{\text{elec}} \cdot e^{\text{l-elec}}_{\text{bit}} + l^{\text{opt}} \cdot e^{\text{l-opt}}_{\text{bit}} + s \cdot e^{s}_{\text{bit}}) \cdot 8 \cdot \text{Data}^{C_n} \\ E^{C_n}_{\text{communication-dynamic}} &= (l^{\text{elec}} \cdot e^{\text{l-elec}}_{\text{bit}} + l^{\text{opt}} \cdot e^{\text{l-opt}}_{\text{bit}} + (l-1) \cdot e^{s}_{\text{bit}}) \cdot 8 \cdot \text{Data}^{C_n} \end{aligned} \tag{6.11}$$

The average number of switches s is straightforward to calculate from the average number of links l that the messages of the communicating pairs have to traverse through the network. The remaining model parameters represent the following. Data^{C_n} represents the total amount of bytes sent by a process of class C_n to all its destination nodes, $e^{\text{l-elec}}_{\text{bit}}$ the energy per bit consumed by the electrical link driver, $e^{\text{l-opt}}_{\text{bit}}$ the energy per bit consumed by the optical link driver and e^{s}_{bit} the energy per bit consumed by the switch logic.

In summary, to estimate the power consumption of the system P_{system}, we use Equation 6.12.

$$P_{\text{system}} = \frac{\sum_{C_n} |C_n| \cdot E^{C_n}_{\text{compute}}}{T_{\text{App}}} + \frac{E_{\text{communication}}}{T_{\text{App}}} \tag{6.12}$$

6.5 Related Work

Many system performance modeling and design-space exploration methods have been proposed in the past. Sharapov et al. [117] propose a solution for capturing the behavior of applications at petascale by combining queuing-based modeling with cycle-accurate simulation. The authors present an infrastructure for manual performance characterization of individual applications via code inspection and benchmarking and use Amdahl's law for predicting performance at petascale. In contrast we propose an automatic way of characterizing parallel applications in a hardware-agnostic manner and, to predict performance, we use hardware analytic models that capture the fundamental interactions between the software properties and the hardware parameters. The only application property that we currently do not automatically detect from PISA profiles is the *type* of the communication pattern (e.g., uniform, shift, nearest-neighbor) which is used on the communication modeling side. This aspect represents a future work item for this thesis.

A large number of previous solutions that aim to model system performance and perform design-space exploration, rely on measurements of existing systems and often involve in-depth knowledge about the applications [133, 81, 64, 120].

Zhai et al. [133] propose a method for predicting behavior of applications at scale. They rely on the existence of at least one node of the target system and predict performance by scaled simulation and replay of the compute and communication graph. A critical limitation of their approach is that the problem size that their solution can deal with is limited by the scale of host platforms from which they collect message logs required for the replay phase.

Snavely et al. [120] use x86-based instrumentation to extract machine profiles using Machine Access Pattern Signature and Pallas MPI benchmarks. To estimate application execution time, they use traces of compute and communication events as input to a network simulator, DIMEMAS [32]. The authors report error rates of less than 10% on average and maximum error rates of approximately 25%. Such an approach has a reasonably good prediction accuracy, but it has poor scalability. Indeed, only the collection and processing of a large-scale communication graph would be problematic or even not possible due to processing or storage limitations. The authors present prediction results for up to 128 nodes.

Gahvari et al. [64] provide a performance model of an algebraic multi-grid kernel, a solver for large sparse linear systems that is used in many HPC applications. Parameters of the analytic performance models are calibrated based on measurements on existing hardware architectures. Moreover expert application knowledge is required to derive application-specific performance models. Similarly, Kerbyson et al. [81] propose a performance model for SAGE, a multi-material hydrodynamics code. With our methodology, we provide a set of more broadly-applicable and hardware-measurement-independent models, and automatic tools for application characterization.

We share with Hoefler et al. [71, 70] the same objectives of early-design-stage exploration of the interaction between applications and compute architectures, in the context of performance and power consumption. In contrast to their approach, however, we do not manually construct analytic models of specific applications. (1) Our compute models are universally applicable given a PISA profile and no previous knowledge of the application compute behavior is required. (2) On the communication modeling side, even though, in the estimation of the communication time, we use similar system parameters, network latency and bandwidth, as in their LogGP models, we do not measure these parameters, but derive them analytically for specific classes of communication patterns, network topologies and mapping strategies.

Using PISA software profiles with analytic models can be compared to other performance evaluation approaches, ranging from theoretical performance models [54] to system simulators and emulators [25, 105, 35]. The latter are capable of generating high-accuracy performance estimates, incurring, however, a high evaluation overhead per hardware specification. The former produce estimates rapidly, but fail to capture details of application behavior as the application is modeled abstractly. We aim to provide a middle ground between the two, by complementing purely analytic performance models with a detailed hardware-independent application model extracted by PISA.

Finally, another related approach uses regression modeling for predicting performance of single-threaded applications [89]. The authors build linear models that predict application performance based on 23 micro-architectural parameters. To build reliable models with such a high number of features, the authors run thousands of simulations using an out-of-order, super-scalar processor simulator. It is reported that the best prediction results are obtained by creating application-specific linear models with median error rates of 4.1% to 10.8%. In our approach, we target early design-space exploration of large-scale systems following two main ideas: (1) decoupling the application properties from the hardware parameters, and (2) deriving hardware analytic models that capture the interaction between the two. We believe that training application-specific performance models (generating training data) for a large number of system parameters (not only processor-related, but also network-related) using simulators is a slow process that we propose to avoid by using hardware analytic models.

6.6 Conclusions

In this chapter we presented the first methodology that estimates the performance of large-scale systems using as input platform-independent software profiles. These profiles were loaded into analytic processor and network models. We evaluated our approach using two applications, Graph 500 and NAS LU, which we ran on systems with wide sets of network configurations.

For the Graph 500 benchmark, we obtained very good correlation results across different hardware systems. This indicates that the proposed methodology could reliably be used to (1) rank systems based on their performance, and (2) perform early and fast design-space exploration. For the NAS LU benchmark, we obtained also good correlation results when using PISA profiles with hardware models. This is again an encouraging result for our approach.

As future work, we will investigate the impact of extrapolation of software profiles on the quality of the performance estimates. We also envision to validate our method with other MPI applications with uniform, shift or 2-dimensional nearest-neighbor communication pattern.

7 Design-Space Exploration Studies in Radio Astronomy and Graph Analytics

In this chapter we apply our performance modeling methodology to two fields: radio astronomy, the Square Kilometer Array [42] and graph analytics, Graph 500 [4]. For the radio astronomy case we perform a design-space exploration study of compute nodes, guiding thus the design of parts of the SKA compute infrastructure.

For the graph analytics case study, we perform a full-system design-space exploration of compute node architectures and network topologies. We identify which combination of processing nodes and network topologies best suit the compute and communication requirements of the most scalable MPI implementation of the Graph 500 benchmark.

7.1 Design-Space Exploration of Compute Nodes

One of the main challenges that the Square Kilometer Array (SKA) telescope will need to surmount is power consumption. In order to make the right choices early in the design process, we analytically study several components of the SKA processing pipeline and estimate their power consumption by assuming different types of compute node architectures. Our objective is not only to provide power consumption estimates for the SKA pipeline, but also to provide guidelines for the SKA compute node architectures. The SKA pipeline will consist of a three-level processing scheme, briefly presented in Subsection 7.1.1. However, for our design-space exploration study we will focus on the first two levels of the pipeline.

In Subsection 7.1.2 we present the first level of the SKA processing pipeline, the station processor. This level will most probably be implemented in ASIC or FPGA technology. Our PISA-based analytic methodology applies only to general-purpose processors, thus cannot be used to estimate the power consumption for ASIC/FPGA technology. Therefore, for this first level of processing, we introduce an application-specific ASIC/FPGA power model. The model

---

This chapter is based on an IEEE conference publication, A. Anghel, R. Jongerius, G. Dittmann, J. Weiss and R. P. Luijten, "Holistic power analysis of implementation alternatives for a very large scale synthesis array with phased array stations," 2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Florence, 2014, pp. 5397-5401, DOI: 10.1109/ICASSP.2014.6854634.

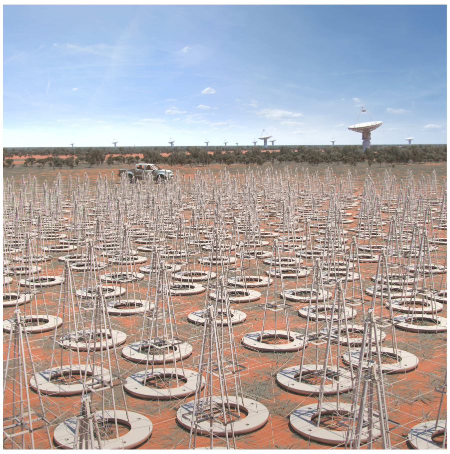


Figure 7.1 – Close-up view of one of the SKA instruments. [From: http://skatelescope.org]

is an extension of the computational requirements analysis proposed by Jongerius [75] for the LOFAR pipeline [56]. Additionally the models herein take into account the power consumption of the memory operations. We also introduce scaling rules for the energy values of arithmetic and memory operations.

In Subsection 7.1.3 we describe the second level of the SKA processing pipeline, the central signal processor. Various compute technologies have been proposed for implementing this SKA processing level, including ASICs, FPGAs and general-purpose CPUs. We analyze the power consumption of the most compute-intensive kernel of this processing level assuming implementations of general-purpose CPUs. As prediction methodology, we employ PISA coupled with analytic processor performance-power models.

7.1.1 Square Kilometer Array Overview

Processing large data volumes in real-time prevents state-of-the-art radio telescopes from achieving the accuracy necessary to study radio signals that originated billions of years ago. The Square Kilometre Array (SKA) [42] is a next-generation telescope which aims to overcome these challenges. By providing an infrastructure that transports and processes data rates in the Pb/s range, SKA will be the largest and most precise radio telescope in the world – at least an order of magnitude better than the current state of the art in terms of sensitivity and survey speed.

As the name suggests SKA will have a collecting area of a million square meters, made up of a vast array of antennas located on two continents, South Africa and Australia (see Figure 7.1). Before being made available to astronomers, the radio signals captured by these antennas will need to be electronically processed. This processing step will face many challenges. Indeed, the aggregated data rate from all the antennas will be in the Pb/s range requiring as many as 10^{18} compute operations/second to process under a very limited power budget.

Relying on Moore's Law may not be enough to ensure that the SKA system will benefit from technology advances that can handle its processing challenges. New hardware-software system design practices are necessary to increase the system performance within a given energy budget. Our proposed analytic methodology can be one such novel design practice to explore large ranges of hardware design points.

The SKA telescope will be built in 2 phases. The construction of the first phase (SKA1) will start in 2018 and end in 2023, whereas the construction of the second phase (SKA2) is projected to start in 2023 and end in 2030. In SKA1 only a part of the SKA telescope will be built in order to serve as a proof-of-concept for testing the feasibility of the telescope. The design of SKA2 is still under discussion at the time of writing this thesis. However, it is expected for the current SKA1 design to significantly extend its collecting area.

SKA1 will consist of three telescopes, each operating in different frequency ranges and serving different sets of science applications, such as planet formation, gravitational waves, cosmic magnetism, dark energy or galaxy evolution. Many SKA1 designs have been proposed during the time when this thesis was conducted. For the remainder of the thesis, we will focus on the re-baselining design proposed in [106] for one of the instruments of the SKA1, the SKA1-Low. This instrument is an array of 131,072 antennas and the largest in terms of number of antennas among the three SKA1 instruments.

The 131,072 antennas of the SKA1-Low telescope, grouped in 512 stations of 256 antennas each, are spread out over an area of 50 km in diameter and operate in the 50 - 350 MHz frequency range. This telescope is fundamentally Fourier synthesis (interferometric) arrays. The main principle of interferometry [113] is to combine the signals from an array of antennas to virtually form the equivalent of a single much larger telescope. For the SKA1-Low, the signals of 256 antennas in a station are forming one station beam and the station beams are combined by correlation. This technology has been proven to be highly flexible at observation time and to operate at a low cost for frequencies lower than 300 MHz [56]. In this work we focus on the digital processing and not on the phase array hardware technologies.

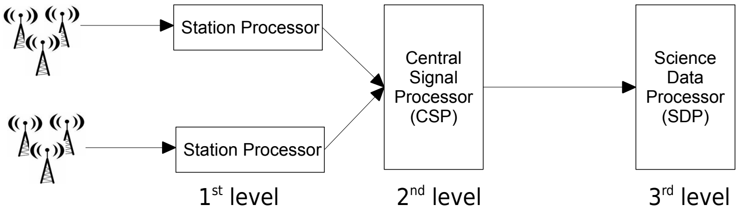


Figure 7.2 – Top-level view of the SKA processing pipeline.

Figure 7.2 shows a top-level view of the processing levels of the SKA pipeline. As a preliminary processing step, the radio signals are filtered and amplified where they are collected at the antenna site. The first processing level happens in a station computing facility (the station processor). 256 antennas send their collected signals to this station processor in order for the signals to be digitized, Fourier transformed and beam-formed. The output of the 512 station processors is sent to a central signal processor (CSP) where the second level of processing occurs. In the CSP, the station beams are correlated with each other. Finally, the correlated frequency channels are sent to yet another computing facility, called the science data processor (SDP), for the third processing step, the sky image processing.

7.1.2 Power Modeling of the SKA Station Processor

For the station processor we estimate the power consumed by the poly-phase filter (PPF) and beam-forming (DBF) kernels. We chose to analyze ASICs and FPGAs, because (1) these are two types of compute nodes that have been often used in the implementation of the digital pipeline of radio telescopes [34] [56], and (2) there is a high probability for the station processor to actually be implemented using ASIC/FPGA technology. With our analysis we aim to provide an optimistic estimate of the compute and memory power requirements of the digital processing pipeline. We also show how an ASIC-based solution compares power-wise to an FPGA-based implementation.

Figure 7.3 shows a schematic view of the station processor, where the system parameter names in the figure are described in Table 7.1. The values of the parameters in the table are in according to the SKA1 re-baseline design [106].

Parameter name	**Parameter description**	**Parameter value**
N_{ant}	no. of antennas (per station)	256 (antennas)
N_{pol}	no. of polarizations (per antenna)	2 (polarizations)
f	antenna sampling frequency	$600 * 10^6$ (samples/second)
N_{taps}	no. of FIR taps (per FIR filter)	8 (taps)
$N_{subbands}$	no. of FIR subbands (per FIR filter)	512 (subbands)
N_{beams}	no. of beams (per station)	1 (beams)

Table 7.1 – SKA1-Low station design parameters.

Each antenna sends its signal to an analog-to-digital converter (ADC) located in the station computing facility. If the signal bandwidth is f_{signal}, the ADC samples the signal at a Nyquist rate of $f = 2 \cdot f_{signal}$ (assuming no over-sampling). The antennas have two polarizations with one ADC per antenna per polarization.

The digitized samples are then passed as real-valued inputs to a poly-phase filter (PPF) that produces complex output samples (real and imaginary) per subband. The PPF consists of a finite impulse response filter bank (FIR) and a Fourier transform (FFT). The real-valued ADC samples are first passed through an FIR filter bank that separates the input signal into multiple subband components $N_{subbands}$, each carrying a single frequency subband of the original signal. The filter multiplies the current sample, as well as a number of recently received samples, with weights that fix the contribution of each input in the final FIR sample. Each multiplication represents a tap filter.

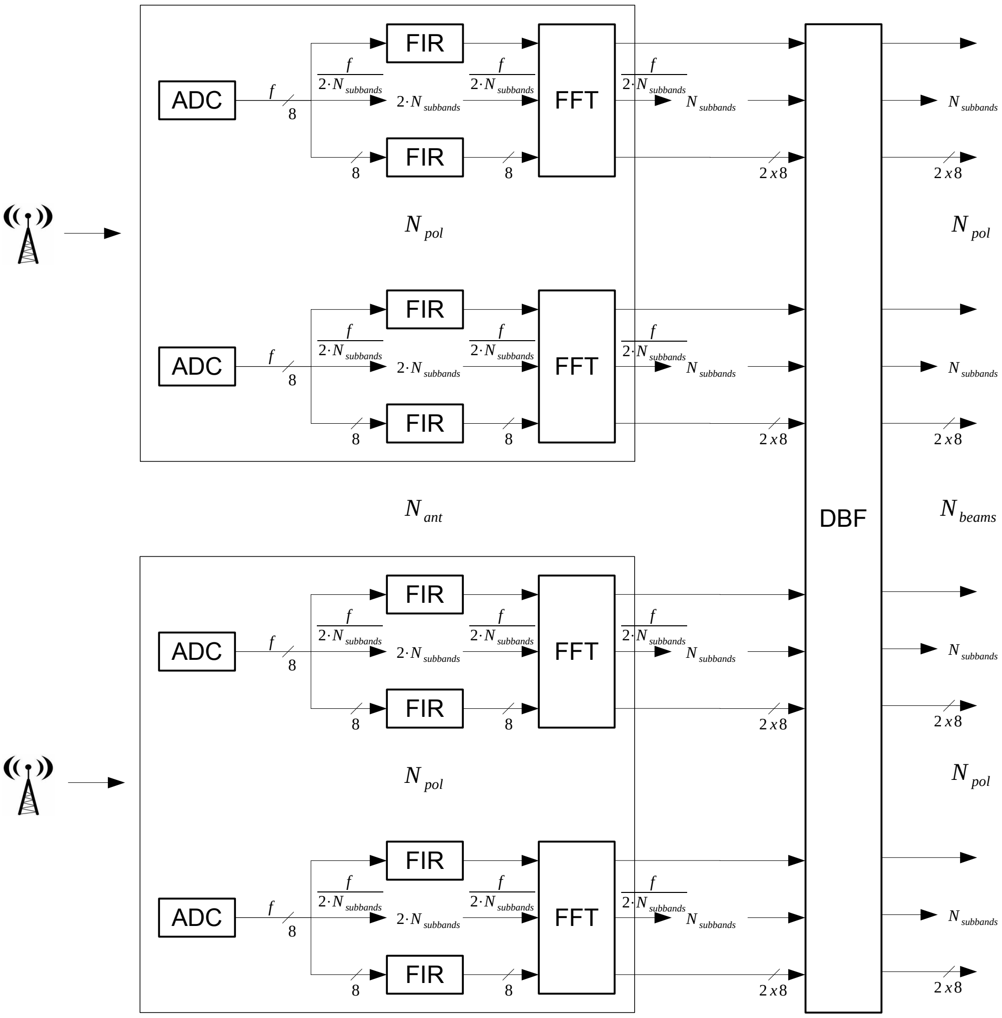


Figure 7.3 – Station digital processing pipeline overview.

The FIR output samples are input to a Fast Fourier Transform (FFT) operation that converts the digitized samples from the time domain into the frequency domain. The FFT size is given by the number of FFT inputs and is typically chosen as a power of 2 to facilitate the addressing in digital storage. Moreover, because the FIR output samples are real-valued and given the Hermitian symmetry of the Fourier transform, two sets of real-valued input sequences can be FFT-processed simultaneously. Thus, the number of subbands $\mathrm{N_{subbands}}$ generated by the real-valued FIR filter is half the number of FFT inputs $\mathrm{N_{fft}} = 2 \cdot \mathrm{N_{subbands}}$.

The FFT output frequencies are input to a digital beam-former (DBF) which applies a phase delay on each frequency sample to steer the beam in a certain direction on the sky and then sums the resulting samples from all antennas per frequency subband. The station beam-former generates $\mathrm{N_{pol}} \cdot \mathrm{N_{subbands}} \cdot \mathrm{N_{beams}}$ streams that are sent to the central processing facility (CSP).

7.1.2.1 ASIC/FPGA Power Modeling Description

Table 7.2 provides a list of power- and energy-related notations used throughout this section.

Notation	Notation description
$\mathrm{P_{FIR}}$	FIR dynamic compute power
$\mathrm{P_{FIR\text{-}MEM}}$	FIR dynamic memory power
$\mathrm{P_{FFT}}$	FFT dynamic compute power
$\mathrm{P_{FFT\text{-}MEM}}$	FFT dynamic memory power
$\mathrm{P_{PPF}}$	FFT+FIR dynamic compute power
$\mathrm{P_{PPF\text{-}MEM}}$	FFT+FIR dynamic and static memory power
$\mathrm{P_{DBF}}$	DBF dynamic compute power
$\mathrm{P_{DBF\text{-}MEM}}$	DBF dynamic and static memory power
$\mathrm{P_{mem}^{leak}}$	Memory leakage power (memory size mem)
$\mathrm{P_{station}}$	Station compute and memory power
$\mathrm{E_{MAC}^{b}}$	Energy per real-valued MAC op. (b-bit operands)
$\mathrm{E_{CMAC}^{b}}$	Energy per complex-valued MAC op. (2 × b-bit operands)
$\mathrm{E_{mem}^{b}}$	Energy per R/W memory op. (b-bit access, mem memory size)
$\mathrm{E_{bfly}^{b}}$	Energy per FFT butterfly operation (2 × b-bit)

Table 7.2 – Power- and energy-related notations and parameters.

The poly-phase filter (PPF) power includes the FIR filtering and FFT processing steps. An $\mathrm{N_{taps}}$ FIR requires $\mathrm{N_{taps}}$ multiply-accumulate (MAC) operations to filter a digital sample generated by the ADC. Equation 7.1 shows the FIR power model. For the SKA1-Low station processing, the FIR input and output operand widths are real-valued 8-bit ($\mathrm{b} = 8$).

$$\mathrm{P_{FIR}} = \mathrm{N_{ant}} \cdot \mathrm{N_{pol}} \cdot \mathrm{N_{taps}} \cdot \mathrm{E_{MAC}^{b}} \cdot \mathrm{f} \tag{7.1}$$

We estimate the number of FFT operations assuming a typical implementation, the radix-2 Cooley-Tukey [Cooley]. This implementation recursively breaks down an $\mathrm{N_{subbands}}$-point discrete Fourier transform into r smaller transforms of size m where r is the radix of the transform and $\mathrm{N_{subbands}} = r \cdot m$. Such a smaller transform operation is called butterfly. We consider the radix r to be equal to 2 and we calculate the total number of FFT compute operations based on the number of butterflies required to process $\mathrm{N_{subbands}}$ inputs. This is $\mathrm{N_{bfly}} = \frac{\mathrm{N_{subbands}}}{2} \cdot \log_2(\mathrm{N_{subbands}})$. Equation 7.2 shows the FFT power model:

$$\mathrm{P_{FFT}} = \mathrm{N_{ant}} \cdot \mathrm{N_{pol}} \cdot \mathrm{N_{bfly}} \cdot \mathrm{E_{bfly}^{b}} \cdot \frac{\mathrm{f}}{2 \cdot \mathrm{N_{subbands}}} = \mathrm{N_{ant}} \cdot \mathrm{N_{pol}} \cdot \log_2(\mathrm{N_{subbands}}) \cdot \mathrm{E_{MAC}^{b}} \cdot \mathrm{f} \tag{7.2}$$

where $\mathrm{E_{bfly}^{b}}$ represents the energy per butterfly operation for an input operand width of $2 \times \mathrm{b}$ bits. Based on its internal real-valued MAC and accumulate (ADD) operation count, a complex-valued butterfly operation requires the same energy as approximately 4 real-valued MAC operations with b-bit inputs. For the SKA1-Low station processor, the inputs to the FFT block are 8-bit real-valued, all the intermediate calculations within the FFT are performed on 2 × 16-

bit operands (16-bit real and 16-bit imaginary) and the outputs are 2 × 8-bit operands (8-bit real and 8-bit imaginary).

It is important to note that the FFT power model is dependent on the assumed FFT implementation. In our model we chose the radix-2 Cooley-Tukey implementation. However, there might be others with less computations, e.g., radix-4 Cooley Tukey. In that case, the user would have to simply replace the number of butterfly operations with the new number (and type) of operations.

By summarizing Equations 7.1 and 7.2, the compute power of the poly-phase filter (PPF) is:

$$P_{PPF} = P_{FIR} + P_{FFT} \tag{7.3}$$

where P_{FIR} is calculated for 8-bit operands (E^{8}_{MAC}) and P_{FFT} is calculated for 2 × 16-bit operands (E^{16}_{MAC}).

We have so far quantified the compute power of the FIR and FFT kernels. However, both kernels use data that resides in memory, thus they generate access operations to read/write data from/to memory. We assume that each PPF has an on-chip memory where the samples in the FIR pre-filters, the FIR coefficients, the FFT inputs and the FFT coefficient (twiddle) factors used in the butterfly calculation are stored. The memory storage size and the memory access size will allow us to quantify the energy consumption of a read/write memory access. Then, based on the number of memory accesses performed by each kernel, we will derive a power model for the memory activity of the FIR and FFT kernels.

Per antenna and polarization, the storage required by the FIR filters is $N_{taps} \cdot 2 \cdot N_{subbands}$ multiplied by the input sample size and that of the FIR coefficients is $N_{taps} \cdot 2 \cdot N_{subbands}$ multiplied by the FIR coefficient size. In our case both input sample size and FIR coefficient size are equal to 8 bits. For each digitized sample and for each FIR filter tap, we assume that at least two reads are performed during filtering: one read for the input sample and one read for the FIR coefficient.

$$P_{FIR\text{-}MEM} = N_{ant} \cdot N_{pol} \cdot N_{taps} \cdot (E^{b^{FIR\text{-}in}}_{mem} + E^{b^{FIR\text{-}coef}}_{mem}) \cdot f \tag{7.4}$$

where $E^{b^{FIR\text{-}in}}_{mem}$ and $E^{b^{FIR\text{-}coef}}_{mem}$ are the dynamic energy consumptions for a memory R/W operation on an FIR input (of bit-width $b^{FIR\text{-}in}$) and on an FIR coefficient factor (of bit-width $b^{FIR\text{-}coef}$), respectively, from/to a memory of size mem.

The FFT storage highly depends on the hardware implementation of the FFT kernel. We assume that at a minimum we need to store the FFT inputs of an FFT and the internal twiddle factors used in the butterfly operations. There are $2 \cdot N_{subbands}$ inputs which require 8 bits of memory each.There are also $N_{subbands} - 1$ twiddle (coefficient) factors of 2 × 16 bit-width each. For each FFT operation we assume that the input samples and the twiddle factors have to be read at least once from memory.

$$\mathrm{P_{FFT\text{-}MEM}} = \mathrm{N_{ant}} \cdot \mathrm{N_{pol}} \cdot (\mathrm{E^{b^{FFT\text{-}in}}_{mem}} + \mathrm{E^{b^{FFT\text{-}coef}}_{mem}}) \cdot \frac{\mathrm{f}}{2} \tag{7.5}$$

where $\mathrm{E^{b^{FFT\text{-}in}}_{mem}}$ and $\mathrm{E^{b^{FFT\text{-}coef}}_{mem}}$ are the dynamic energy consumptions for a memory R/W operation on an FFT input (of bit-width $\mathrm{b^{FFT\text{-}in}}$) and FFT coefficient factor (of bit-width $\mathrm{b^{FFT\text{-}coef}}$), respectively, from/to a memory of size mem.

In summary, the PPF power model for data movement through memory is shown in Equation 7.6.

$$\mathrm{P_{PPF\text{-}MEM}} = \mathrm{P_{FIR\text{-}MEM}} + \mathrm{P_{FFT\text{-}MEM}} + \mathrm{N_{ant}} \cdot \mathrm{N_{pol}} \cdot \mathrm{P^{leak}_{mem}} \tag{7.6}$$

where the memory is shared by both the FIR and the FFT and is of size:

$$\begin{aligned}\mathrm{mem} =& \mathrm{N_{taps}} \cdot 2 \cdot \mathrm{N_{subbands}} \cdot (\mathrm{b^{FIR\text{-}in}} + \mathrm{b^{FIR\text{-}coef}}) \\ &+ 2 \cdot \mathrm{N_{subbands}} \cdot \mathrm{b^{FFT\text{-}in}} + (\mathrm{N_{subbands}} - 1) \cdot \mathrm{b^{FFT\text{-}coef}}\end{aligned}$$

and $\mathrm{P^{leak}_{mem}}$ represents the leakage power of this on-chip memory of size mem. The FIR memory accesses are on 8-bit operands and the FFT memory accesses are on 8-bit operands for the inputs and on 2 × 16-bit operands for the FFT coefficient factors.

In the next processing step, the digital beam-former (DBF) applies a phase delay on each FFT subband to steer the beam in a certain direction on the sky and then sums the resulting samples from all antennas per subband. Each complex stream from the FFT output is multiplied by a complex exponential to adjust its phase. Then, all the streams of the same polarizations from different antennas are summed.

The phase delay step can be implemented with one complex multiplication. A complex addition is further required to add in the partial sum from the previous antenna. Thus, each beam-former operation requires one complex-valued MAC (CMAC) per antenna per polarization. This computation occurs independently for each station beam. The energy of a 2 × b-bit CMAC is estimated to consume as much energy as approximately 4 b-bit MACs.

$$\mathrm{P_{DBF}} = \mathrm{N_{beams}} \cdot \mathrm{N_{pol}} \cdot \mathrm{N_{ant}} \cdot 4 \cdot \mathrm{E^{b}_{MAC}} \cdot \frac{\mathrm{f}}{2} \tag{7.7}$$

where the MAC arithmetic is performed on 8-bit operands ($\mathrm{E^{8}_{MAC}}$).

For the beam-forming step, we neglect the source of the delay values. Normally, these values will be computed periodically and stored in a memory of at least length $\mathrm{N_{beams}} \cdot \mathrm{N_{subbands}} \cdot \mathrm{N_{ant}} \cdot \mathrm{N_{pol}}$. So each beam-forming operation requires not only one CMAC, but also one read operation from this memory.

$$P_{\text{DBF-MEM}} = N_{\text{beams}} \cdot N_{\text{pol}} \cdot N_{\text{ant}} \cdot E^{b}_{\text{mem}} \cdot \frac{f}{2} + P^{\text{leak}}_{\text{mem}} \tag{7.8}$$

where an on-chip memory is dedicated to the DBF kernel and is of size:

$$\text{mem} = N_{\text{beams}} \cdot N_{\text{subbands}} \cdot N_{\text{ant}} \cdot N_{\text{pol}} \cdot b^{\text{DBF-in}}$$

where $b^{\text{DBF-in}}$ is the DBF input bit-width (2×8) and $P^{\text{leak}}_{\text{mem}}$ represents the leakage power of this on-chip memory of size mem.

In summary we estimate the power consumption of the station processor as:

$$P_{\text{station}} = P_{\text{PPF}} + P_{\text{PPF-MEM}} + P_{\text{DBF}} + P_{\text{DBF-MEM}} \tag{7.9}$$

7.1.2.2 Power Models Parameters and Scaling Rules

We analyze the power consumption of the digital processing pipeline implemented in ASIC and FPGA. We estimate the dynamic power requirements when implemented in 90 nm technology and extrapolate the power consumption towards a 14 nm technology.

Simulations on synthesized designs prior to layout (22 nm technology, 125 MHz clock) yield an ASIC MAC energy consumption of 9.6 pJ for real-valued 32-bit operands. This value is scaled to 90 nm and 14 nm technologies using the NMOSFET dynamic power indicator values $(C \cdot V^2)$ reported in the high-performance logic technology requirements of the ITRS PIDS tables [6]. The scaling factors are 0.83 for 22 nm → 14 nm and 1.86 for 22 nm → 90 nm transition, where the MAC energy scales quadratically with the bit width of the input operands.

We scale the energy consumption of FPGA MAC operations in 90 nm based on the FPGA vs. ASIC dynamic power measurements reported in [85]. They compare 90-nm CMOS FPGA and 90-nm CMOS standard-cell ASIC in terms of power consumption for core logic. The dynamic power ratio is approximately 7x for FPGAs that use *hard-wired* building blocks (memories, multipliers, DSP) compared to ASICs. In our analysis, we employ the latter value. When moving to 14 nm technology, we consider this relative consumption factor to remain unchanged. However, given the increasing number of hard blocks hosted by FPGAs at lower technology nodes, it is possible that this ratio will decrease. Table 7.3 shows the MAC-related energies used in our models.

For on-chip memories, we assume embedded DRAM for ASICs and SRAM for FPGAs. To estimate the dynamic energy of a R/W memory operation and the memory leakage power, we leverage the CACTI tool [126] (the "pure RAM interface") with the following parameters.

For DRAM we use: LP-DRAM for the RAM cell type in both the data and tag arrays, ITRS-HP for the peripheral and global circuitry transistor type in both the data and tag arrays, conservative

Platform	**ASIC**			**FPGA**			**Eq. no.**
Technology node	90 nm	22 nm	14 nm	90 nm	22 nm	14 nm	
E^{8}_{MAC} (pJ)	1.1	0.6	0.5	7.7	4.2	3.4	7.1, 7.7
E^{16}_{MAC} (pJ)	4.4	2.4	2	31.1	16.8	13.9	7.2
E^{32}_{MAC} (pJ)	17.8	9.6	7.9	124.6	67.2	55.7	-

Table 7.3 – MAC energy consumption values for ASIC and FPGA technology.

interconnect projection type, global type of wire outside mat, temperature of 350 K, 1 read port and 1 write port. For SRAM we use: ITRS-HP for the RAM cell type in both the data and tag arrays, ITRS-HP for the peripheral and global circuitry transistor type in both the data and tag arrays, conservative interconnect projection type, global type of wire outside mat, temperature of 350 K, 1 read port and 1 write port.

For both DRAM and SRAM memories, we assume memory banks of 16 KB each and number of bits read out per memory access of 8-bit. A 16-bit memory access is estimated to consume twice the energy of an 8-bit access. Also the power leakage of an N-bank memory is calculated as the number of banks N multiplied by the power leakage per bank. The memory size of a station PPF is 19,456 bytes (thus two memory banks of 16 KB each summing up to a total memory of 32 KB). The memory size of a station DBF is 524,288 bytes (thus 32 memory banks of 16 KB each summing up to a total memory of 524 KB).

The newest technology supported by CACTI is 32 nm. Thus, for 14 nm technology, we first calculate the R/W dynamic energy for 32 nm using CACTI and then scale it to 14 nm using a factor of 0.8 [6]. As for the leakage power, we assume the scaling factor to 14 nm technology to be 1x. The memory-related energies that we used in our models are shown in Table 7.4 for ASIC technology and in Table 7.5 for FPGA technology.

	Technology node	90 **nm**	32 **nm**	22-**nm**	14 **nm**	**Eq. no.**
PPF	E^{8}_{32kB} (pJ)	19	4.3	4.1	3.4	7.4,7.5
	E^{32}_{32kB} (pJ)	76	17.2	16.5	13.6	7.5
	P^{leak}_{32kB} (mW)	1.3	1.4	1.4	1.4	7.6
DBF	E^{16}_{524kB} (pJ)	78	15.6	14.9	12.4	7.8
	P^{leak}_{524kB} (mW)	48	48	48	48	7.8

Table 7.4 – DRAM access energy consumption values for the ASIC memory model.

7.1.2.3 Results and Discussion

Figure 7.4 shows the per-station power results for ASIC and FPGA implementations in 90, 32 and 14 nm CMOS technologies. The results provide an optimistic estimate of the station power consumption, covering only part of what in reality consumes power in the station.

	Technology node	90 nm	32 nm	22-nm	14 nm	Eq. no.
PPF	E^{8}_{32kB} (pJ)	18	3.2	3.1	2.5	7.4,7.5
	E^{32}_{32kB} (pJ)	72	12.8	12.2	10.2	7.5
	P^{leak}_{32kB} (mW)	14.4	12.4	12.4	12.4	7.6
DBF	E^{16}_{524kB} (pJ)	98	18.6	17.8	14.8	7.8
	P^{leak}_{524kB} (mW)	240	233	233	233	7.8

Table 7.5 – SRAM access energy consumption values for the FPGA memory model.

Components such as analog-to-digital conversion, chip-to-chip communication, static logic power, static refresh memory power or cooling are not part of the model. Also calibration may periodically occur in the digital pipeline that we do not consider in the model.

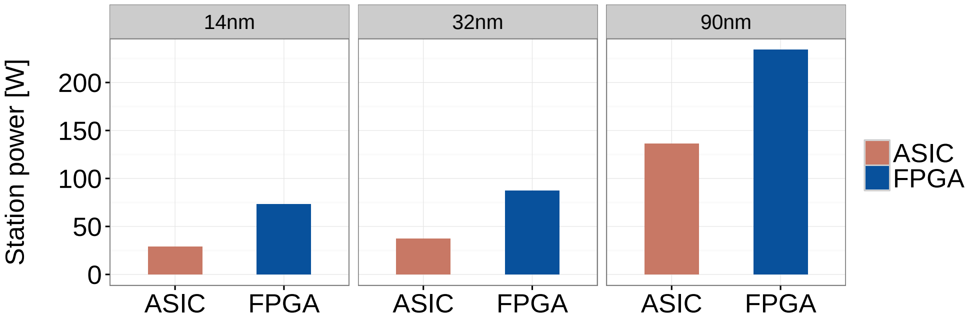


Figure 7.4 – Station processor power for ASIC and FPGA technologies.

The results show that a station would consume at least 29 W and 73 W for ASIC and FPGA implementations in 14 nm, respectively. This implies that (1) all the SKA1-Low 512 stations would consume at least approximately 14.8 kW in ASIC and 37.3 kW in FPGA, and (2) an ASIC-based implementation would be 2.5 times more power-efficient than an FPGA implementation. In 32 nm, a station would consume at least 37 W and 87 W for ASIC and FPGA implementations, respectively. This implies that all the SKA1-Low 512 stations would consume at least approximately 19 kW in ASIC and 44.5 kW in FPGA.

7.1.3 Power Modeling of the Central Signal Processor

The central signal processor collects the data from the $N_{stat} = 512$ station processors and correlates it. The CSP inputs are the beams per subband and antenna polarization generated by each station processor. The total number of inputs is $N_{beams} \cdot N_{pol} \cdot N_{subbands} \cdot N_{stat}$.

The CSP inputs first pass through a second poly-phase filter. This filter further splits the station subbands into narrower frequency ranges, called channels. In the station the original signal was decimated into $N_{subbands} = 512$ subbands. In the CSP each subband is further decimated into $N_{channels} = 128$ channels each, resulting in 65,536 total number of channels, as stated in the SKA1 re-baseline design. All the channels are then input to two additional computational steps, a phase delay and a bandpass correction, both implemented with complex multiplications. The results are further input to the most compute-intensive kernel of this pipeline, the correlation [128]. In this section we will focus on the analysis of this kernel.

The channels are correlated for each pair of stations and polarizations per station beam. For each channel and each combination of two stations and two polarizations, the complex sample from one station is multiplied with the conjugate of the complex sample from the other station. Thus, a correlation is essentially a complex multiply operation (in the case of the SKA1-Low, performed on 2 × 32-bit operands).

To reduce the amount of data generated by the correlation step, an additional integration step is performed immediately after correlation. The integrator accumulates data over a specified time interval and stores the intermediate results. Each correlation is thus followed by a complex addition. Therefore, in terms of compute, the correlator and the integrator perform one CMAC operation per correlation. The total number of correlation-integration operations is $\mathrm{N}_{\mathrm{stat}}^2 \cdot \mathrm{N}_{\mathrm{pol}}^2 \cdot \mathrm{N}_{\mathrm{beams}} \cdot \frac{\mathrm{f}}{2}$. For the SKA1-Low CSP the number of correlations is $314 \cdot 10^{12}$, where $\mathrm{N}_{\mathrm{stat}} = 512$. A back-of-the-envelope power estimate for the compute part of the correlation for ASIC technology in 14-nm is $7.9 \cdot 10^{-12} \cdot 314 \cdot 10^{12} = 2.48$ kW, whereas for FPGA technology is $55.7 \cdot 10^{-12} \cdot 314 \cdot 10^{12} = 17.48$ kW.

However, in addition to the CMAC operations, the correlator and the integrator also perform a large number of memory operations. To efficiently implement the integration in terms of storage size and number of memory accesses, the correlated outputs need to arrive at the integrator in a specific order. For instance, if the correlator receives the samples on a per-channel basis, the correlated outputs will need to be re-ordered on a per-polarization basis and all the samples that belong to the same integration interval will need to arrive back-to-back at the integrator input. A correlator-integrator power model for the memory requirements of an ASIC/FPGA implementation would highly depend on the assumed hardware architecture.

Various ASIC/FPGA correlator designs have been proposed in the literature. For example, Fiorin et al. [62] proposes an ASIC design with programmable, near-data accelerator using 3D-stacked hybrid memory cubes (HMCs). The correlator algorithm is mapped to the proposed architecture and a power model is derived for 14-nm CMOS technology. The paper reports a power consumption of 9.62 W for processing all channels of a station subband, resulting in a total of approximately 5 kW.

Moreover, Romila [114] proposes an FPGA design implemented on a Xilinx Virtex-6 FPGA. The dynamic power of the correlator (including the integration kernel) is reported to be approximately 30 kW, for a slightly different SKA1-Low design point with 512 instead of 128 channels and with a 4 times smaller integration time than the actual one. A detailed analysis of the memory requirements of these proposed architectures or another correlator design are out-of-the-scope of this thesis. We proceed with the analysis of the kernel assuming general-purpose CPUs with traditional cache hierarchy and external DRAM memory. For this analysis we employ our full-system performance prediction methodology presented in Chapter 6.

7.1.3.1 General-Purpose CPU Power Modeling Overview

As input to our methodology we used an OpenMP correlator implementation similar to the one presented in [128]. As the implementation is in OpenMP, the analysis in this section will be restricted to the compute processing (no analysis of the communication over network, for which it would be required an MPI implementation). To characterize the software we used LLVM 3.4 with the `-O3 -mllvm -force-vector-width=16` compiler optimizations. We characterized the software with PISA in terms of instruction mix, instruction-level parallelism, branch behavior and memory temporal reuse patterns, for a set of problem sizes. A problem size is defined by the number of stations, the number of channels and the number of OpenMP threads. Table 7.6 shows the problem sizes that we profiled using PISA.

Parameter	**Train set**	**Test set**
Stations	16,24,40,56,64	56,80,120,160,184
Channels	8,16,32,64,128	32,64,128,256,512
Thread count	2,4,6,8,10	12,14,16,18,20

Table 7.6 – Problem sizes of the correlator implementation profiled with PISA.

The SKA1-Low target size is defined for 512 stations and 65,536 channels. We use the PISA profiles of smaller problem sizes as input to the ExtraX extrapolation framework to estimate the software properties for the target size. We train the machine learning methods used by ExtraX with the PISA results of the problem sizes in the train set. We then evaluate the accuracy of the models on the results obtained for the problem sizes in the test set.

PISA extracts software profiles for each OpenMP thread individually. Before training the models, ExtraX runs a thread clustering method that identifies similarities across threads in the PISA profiles. For the correlation algorithm, the clustering identifies that all worker threads have similar properties, thus the correlator implementation under study exhibits one class of threads. For a given problem size, the profile representative of this class is determined by averaging the workload properties across all threads. The resulting class profiles obtained for the problem sizes in the train set are used in the remainder of the ExtraX analysis to build models that will extrapolate the workload properties to the target scale. For each workload property ExtraX derives a separate extrapolation model.

We evaluate the accuracy of the models on the profiles obtained for the problem sizes in the test set shown in Table 7.6. For all metrics, e.g., instruction count, instruction count per type (integer, floating-point, memory, control), instruction-level parallelism, instruction-level parallelism per type, we obtain accuracies of more than 98%. This means that, for the correlator implementation under study, ExtraX is able to accurately estimate the workload properties at large non-profiled problem sizes.

We use the extrapolation models to derive the workload properties of the threads executing the correlator for the SKA target size. Furthermore, we input the extrapolated profile to a processor hardware model to evaluate the compute performance. We use the compute model in [76] that was partially presented in Section 4.2.3. For the multi-threaded implementations, the model assumes that each OpenMP thread in the application is assigned to one processor core. In our design-space exploration exercise, an n-core processor will run an n-thread correlator. We first evaluate the performance (processor time) for compute nodes representatives of big and small cores, Intel Xeon E5-2697 v3 Haswell-EP and ARM Cortex-A15 (ARM) processors. The hardware parameters of these architectures are presented in Table 7.7.

Parameter	**Description**	**Xeon E5-2697**	**ARM Cortex-A15**
n_{cores}	Cores per socket	14	4
f_{core}	Core clock frequency	2.6 GHz	2.3 GHz
$n^{\text{issue-width}}$	Issue width	8	8
n^{INT}	# integer units	4	3
n^{FP}	# floating-point units	2	2
n^{MEM}	# load/store units	2	2
n^{CTRL}	# branch units	2	1
$n^{\text{front-pipe}}$	Front-end pipeline depth	7	12
n^{ROB}	Reorder buffer capacity	192	128
$M^{\text{d-L1}}$	Data L1 cache size	32 KB	32 KB
$M^{\text{d-L2}}$	Data L2 cache size	256 KB	0 KB
$M^{\text{d-L3}}$	Data L3 cache size	32 MB	2 MB
M^{DRAM}	DRAM size per socket	32 GB	2 GB
$T^{\text{core-L1}}$	Data L1 hit latency	4 cycles	4 cycles
$T^{\text{core-L2}}$	Data L2 hit latency	12 cycles	0 cycles
$T^{\text{core-L3}}$	Data L3 hit latency	36 cycles	21 cycles
$T^{\text{core-DRAM}}$	DRAM hit latency	217 cycles	274 cycles
B^{L1}	L1 cache bandwidth	160 GB/s	137 GB/s
B^{L2}	L2 cache bandwidth	160 GB/s	137 GB/s
B^{L3}	L3 cache bandwidth	40 GB/s	17 GB/s
B^{DRAM}	DRAM bandwidth	59.7 GB/s	17 GB/s

Table 7.7 – Values for hardware parameters of Intel Xeon E5-2697 v3 and ARM Cortex-A15

On the application side, each channel can be processed independently from any other. Therefore, we assume that each compute node processes an equal set of channels from the total of 65,536. We assume 8 channels and 512 stations per compute node. The analytic processor performance model used with the PISA profile extrapolated to 512 stations for 14 threads (Haswell) and 4 threads (ARM), reports the following performance numbers. A compute node requires 5.42 seconds of processing on a Haswell processor and 12.79 seconds on an ARM Cortex-A15 processor. These execution times violate the real-time requirement imposed by the system, 0.52 seconds (the CSP integration time). Therefore, we further explore variations of the two architectures for several hardware parameters as shown in Table 7.8 in order to determine a processor architecture that could meet the time requirement.

Parameter	**Values**	**Unit**
Core count n_{cores}	48,64,96,128	cores
Clock frequency f_{core}	2.3,2.6,3.2,3.6,4.0	GHz
L2-L3 memory bandwidth B^{L3}	40,60,120,160	GB/s
L3-DRAM memory bandwidth B^{DRAM}	40,60,120,160	GB/s

Table 7.8 – Design-space exploration of processor parameters.

7.1.3.2 Results and Discussion

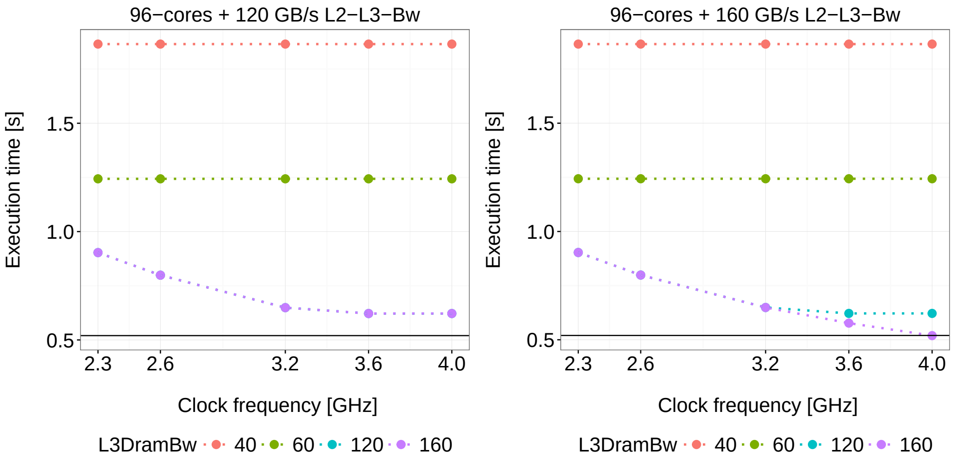
In the remainder of this section, the performance metric is defined as the execution time of a compute node that runs the correlation kernel. Also the power metric is the power consumed by a compute node (processor and memory) during the execution of the correlation.

We start our analysis for Haswell architectures assuming that a compute node processes 8 channels and 512 stations. We first investigate which Haswell processor architectures meet the time requirement of 0.52 seconds. We identify that none of the architectures in our hardware design space with 48 and 64 cores per processor and with L2-L3 memory bandwidths of 40 GB/s and 60 GB/s meets the time constraint. Therefore, we show in Figures 7.5 and 7.6 only the exploration results for architectures with a number of cores of 96 and 128 and for L2-L3 memory bandwidths of 120 GB/s and 160 GB/s.

Each of the graphs in Figures 7.5 and 7.6 presents the results for a certain combination of number of cores and L2-L3 memory bandwidth for all the clock frequencies and L3-DRAM memory bandwidths in our hardware design space. Figure 7.5 shows the results for 120 GB/s and 160 GB/s L2-L3 memory bandwidth, for 96 cores. Figure 7.6 shows the results for 120 GB/s and 160 GB/s L2-L3 memory bandwidth, for 128 cores. The horizontal black line in all sub-figures represents the time requirement imposed by the system.

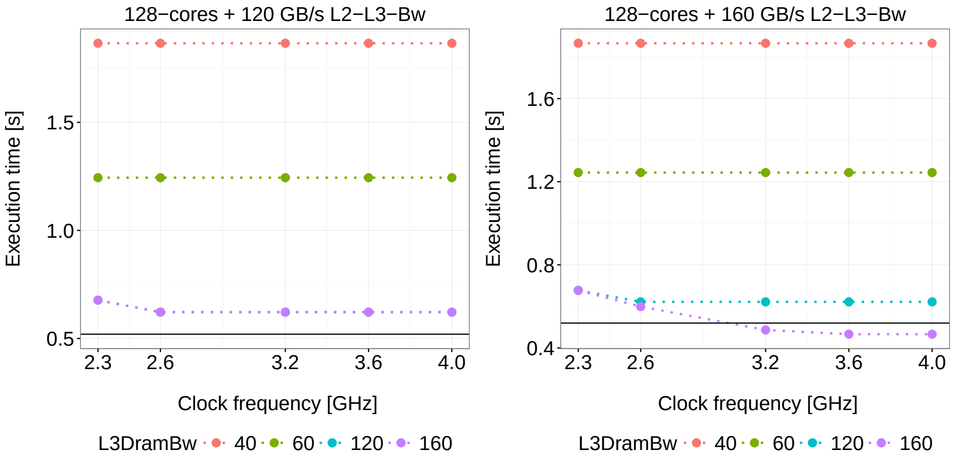
Generally, we observe from each of the graphs that the higher the L3-DRAM memory bandwidth, the lower the execution time. Indeed the lowest execution times are of approximately 0.5 seconds obtained for an L3-DRAM memory bandwidth of 160 GB/s.

For L3-DRAM memory bandwidths of 40 GB/s and 60 GB/s, the clock frequency does not impact the execution time, most probably because the system is memory bandwidth bound. However, for an L3-DRAM memory bandwidth of 120 GB/s, the clock frequency impacts the



(a) 120 GB/s L2-L3 memory bandwidth (b) 160 GB/s L2-L3 memory bandwidth

Figure 7.5 – CSP design-space exploration of Haswell nodes – 8 channels per node, 96 cores.



(a) 120 GB/s L2-L3 memory bandwidth (b) 160 GB/s L2-L3 memory bandwidth

Figure 7.6 – CSP design-space exploration of Haswell nodes – 8 channels per node, 128 cores.

execution time, especially in the case of 96 cores (Figure 7.5(a)) where the execution times decreases with approximately 25% when increasing the clock frequency from 2.6 GHz to 4.0 GHz. Nevertheless, none of the 96 cores, 120 GB/s L2-L3 memory bandwidth architectures meets the time requirement of 0.52 seconds. The same applies for the 128 cores, 120 GB/s L2-L3 memory bandwidth architectures (Figure 7.6(a)).

The only processor configurations that meet the time constraint imposed by the system are shown in Figures 7.5(b) and 7.6(b): (i) (Figure 7.5(b)) 96 cores, 160 GB/s L2-L3 memory bandwidth, 160 GB/s L3-DRAM memory bandwidth for clock frequency of 4.0 GHz, and (ii) (Figure 7.6(b)) 128 cores, 160 GB/s L2-L3 memory bandwidth, 160 GB/s L3-DRAM memory

Cores	Frequency	L2-L3 b/w	L3-DRAM b/w	Time	Power	Total power
96	4.0	160 GB/s	160 GB/s	0.51 s	421 W	3.44 MW
128	3.2	160 GB/s	160 GB/s	0.48 s	511 W	4.18 MW
128	3.6	160 GB/s	160 GB/s	0.46 s	521 W	4.26 MW
128	4.0	160 GB/s	160 GB/s	0.46 s	521 W	4.26 MW

Table 7.9 – Performance-optimal Haswell architectures for 8 channels per node.

bandwidth for clock frequencies of 3.2, 3.6 and 4.0 GHz. For these configurations, we perform a power-performance trade-off analysis. Table 7.9 shows the power-performance results for these 4 architecture designs. The total power is calculated by multiplying the power of a node by $8{,}192 = \frac{65{,}536 \text{ channels}}{8 \text{ channels per node}}$.

Out of the four processor architectures in Table 7.9 the one with the lowest power consumption is the with 96 cores. In this case, the total power consumption of the correlator is 3.44 MW.

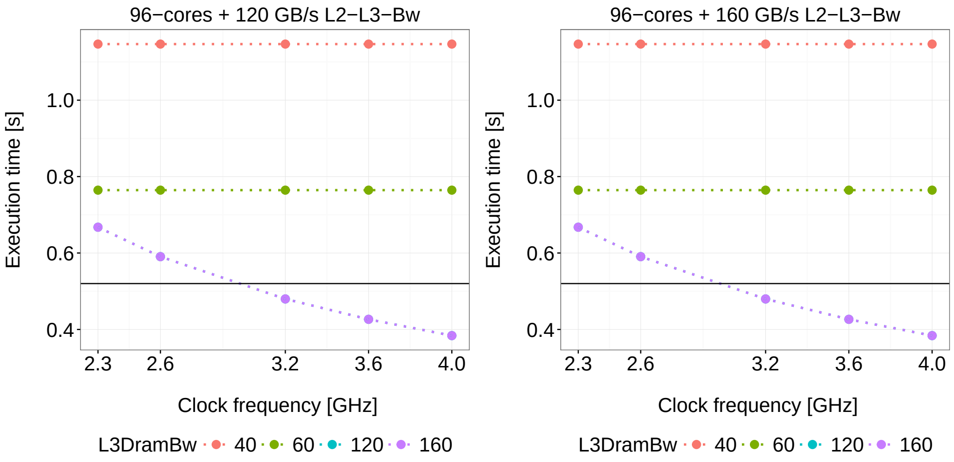
We performed the same design-space exploration for ARM processor architectures. However, none of the hardware design points in Table 7.8 was able to run the correlation kernel for 8 channels and 512 stations in less than 0.52 seconds. Indeed, the fastest hardware design point was an ARM-type of processor with 128 cores, 160 GB/s L2-L3 memory bandwidth, 160 GB/s L3-DRAM memory bandwidth and 4.0 GHz clock frequency. The execution time of this architecture was 0.57 seconds with 364 W per compute node, summing up to a total of 2.98 MW. We further analyze the ARM architectures assuming 4 channels and 512 stations per compute node, as less workload per compute node will most probably allow us to find architectures that meet the time constraint (same hardware design space as in Table 7.8).

As in the case of Haswell, we first investigate which ARM processor architectures meet the time requirement of 0.52 seconds. We identify that none of the architectures in our hardware design space with 48 and 64 cores per processor and with L2-L3 memory bandwidths of 40 GB/s and 60 GB/s meets the time requirement. Therefore, we show in Figures 7.7 and 7.8 only the exploration results for architectures with a number of cores of 96 and 128 and for L2-L3 memory bandwidths of 120 GB/s and 160 GB/s.

Each of the four graphs in Figures 7.7 and 7.8 presents the results for a certain combination of number of cores and L2-L3 memory bandwidth for all the clock frequencies and L3-DRAM memory bandwidths in our hardware design space. Figure 7.7 shows the results for 120 GB/s and 160 GB/s L2-L3 memory bandwidth, for 96 cores. Figure 7.8 shows the results for 120 GB/s and 160 GB/s L2-L3 memory bandwidth, for 128 cores.

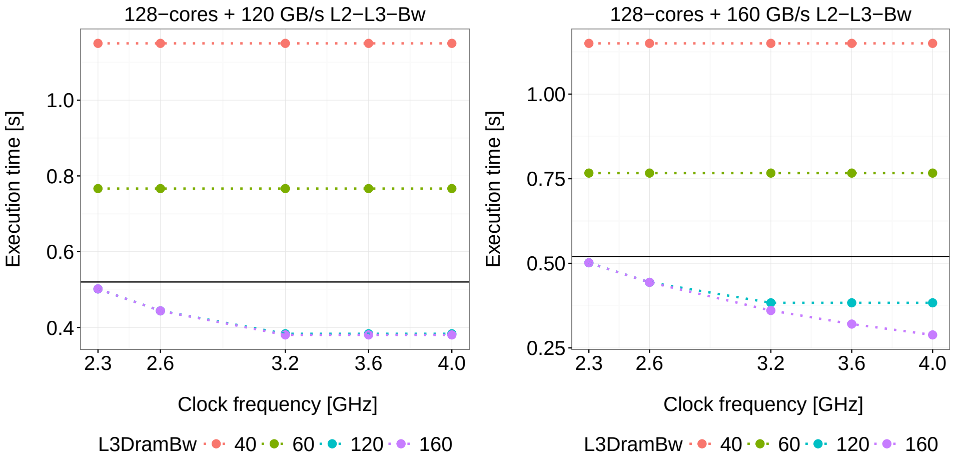
Generally, we observe from each of the graphs that the higher the L3-DRAM memory bandwidth, the lower the execution time. Indeed, the lowest execution times are obtained for an L3-DRAM memory bandwidth of 160 GB/s.

For L3-DRAM memory bandwidths of 40 GB/s and 60 GB/s, the clock frequency does not impact the execution time, most probably because the system is memory bandwidth bound.



(a) 120 GB/s L2-L3 memory bandwidth (b) 160 GB/s L2-L3 memory bandwidth

Figure 7.7 – CSP design-space exploration of ARM nodes – 4 channels per node, 96 cores.



(a) 120 GB/s L2-L3 memory bandwidth (b) 160 GB/s L2-L3 memory bandwidth

Figure 7.8 – CSP design-space exploration of ARM nodes – 4 channels per node, 128 cores.

However, for an L3-DRAM memory bandwidth of 120 GB/s or 160 GB/s, the clock frequency impacts the execution time, as the system becomes compute bound. Indeed, in the case of 96 cores (Figure 7.7(a)) the execution time decreases with approximately 30% when increasing the clock frequency from 2.6 GHz to 4.0 GHz. Similar conclusions can be drawn from the other results for 96 cores and 128 cores.

Overall, we identify 32 design points that meet the time requirement imposed by the system, time lower than or equal to 0.52 seconds. For 96 cores (Figure 7.7), regardless of whether the L2-L3 and L3-DRAM memory bandwidths are 120 or 160 GB/s, a clock frequency of 3.2 GHz gives an execution time of approximately 0.47 seconds under a power consumption of 255 W per node (summing up to a total power consumption of 4.17 MW for 16,384 = $\frac{65{,}536 \text{ channels}}{4 \text{ channels per node}}$ compute nodes). The hardware designs with higher clock frequencies of 3.6 or 4.0 GHz show a higher power consumption by up to 5% in comparison with the frequency of 3.2 GHz. For 128 cores (Figure 7.8), any of the clock frequencies gives a design point that meets the time constraint. However, the lowest power consumption across the 128-core designs that meets the time target is 318 W, which is 24% higher than the optimal 96-core design previously identified, making it sub-optimal.

In summary, we have identified two systems that could implement the SKA correlation under the time constraint of 0.52 seconds: (1) a system with 8192 compute nodes, each node with a Haswell-type of processor architecture with 96 cores, 160 GB/s L2-L3 memory bandwidth, 160 GB/s L3-DRAM memory bandwidth and frequency of 4.0 GHz, and (2) a system with 16384 compute nodes, each node with an ARM-type of processor architecture with 96 cores, 120 GB/s L2-L3 memory bandwidth, 120 GB/s L3-DRAM memory bandwidth and frequency of 3.2 GHz. Although ARM processors are known to be less power-hungry than Haswell processors, in this case, the ARM solution, given the higher number of nodes necessary to meet the time requirement, consumes 20% more power than the Haswell-based solution. We conclude that the Haswell-based solution is a possible system for implementing the SKA correlation under a power budget of 3.44 MW.

7.2 Design-Space Exploration of Large-Scale Systems

In this subsection we explore hardware designs of compute nodes and network topologies and analyze their performance and power consumption for the most scalable MPI implementation (MPI-simple) of Graph 500 [4]. The performance metric is the application execution time, comprising of the compute and communication times. The time of an MPI application is analytically estimated as described in Chapter 6. The power consumption of the system is calculated based on the power consumed by the compute nodes and network. The power of the compute nodes is calculated as described in Chapter 4 and includes both the processor and the DRAM power. To estimate the power consumption of the entire system (network included) we use the model described in Section 6.4.

As input to our methodology we use the MPI-simple implementation of Graph 500. To characterize the software we use LLVM 3.4 with the `-O3 -mllvm -force-vector-width=16` compiler optimizations. We characterize the software with PISA in terms of instruction mix, instruction-level parallelism, memory temporal reuse patterns and communication pattern, for a set of problem sizes (no branch behavior analysis). A problem size is defined by the scale and edge factor of the graph and the number of MPI processes. Table 7.10 shows the problem sizes that we profiled with PISA.

We define the target size to be for a scale of 26, edge factor of 16 and 256,144 MPI processes. We use the PISA profiles of smaller problem sizes as input to the ExtraX extrapolation framework to estimate the software properties for the target size. ExtraX trains machine learning models

Parameter	Train set	Test set
Scale	14,15,16,17	21,22,23
Edge factor	16,32,64	32,64,128
MPI processes count	4,8,16	32,64,128

Table 7.10 – Problem sizes of the Graph 500 MPI-simple code profiled with PISA.

using the PISA profiles of the problem sizes in the train set. We then evaluate the prediction accuracy of the models on the results obtained for the problem sizes in the test set.

PISA extracts software profiles for each individual MPI process. Before training the models, ExtraX runs a process clustering method that identifies similarities across processes in the PISA profiles. For the Graph 500 BFS kernel, the clustering identifies that all worker processes have similar properties, thus the BFS implementation under study exhibits a single class of processes. For a given problem size, the profile representative of this class is determined by averaging the workload properties across all processes. The resulting class profile is used in the remainder of the ExtraX analysis to build models that will extrapolate the workload properties to the target scale. For each workload property ExtraX derives a separate extrapolation model.

We evaluate the accuracy of the models on the profiles obtained for the problem sizes in the test set shown in Table 7.10. The average prediction rate for the instruction count is 4.7%, the average prediction rate of the memory reuse distance of 1.4%, the average prediction rate for the overall ILP of 0.7% and the average prediction rate for the number of exchanged MPI messages of 9.7%.

We use the extrapolation models to derive the workload properties of the processes executing the BFS kernel of the Graph 500 benchmark at target scale. Finally, we input the extrapolated profile to a processor hardware model to evaluate the compute performance. We use the compute model in [76] that was partially presented in Section 4.2.3. For estimating the network performance we use the uniform bandwidth models for fat-tree, torus and 2D HyperX topologies in Section 5.4.

We perform an exploration study across different processors and network topologies for a system scale of 262,144 compute nodes. This study could be of interest for designers of large-scale systems that are expected to run Big Data workloads such as graph analytics. As compute nodes, we use E5-2697 v3 (Haswell) and Tegra K1 ARM Cortex-A15 processors (Table 7.7). As network topologies, we consider variations of 3-level fat-tree, 2D HyperX and 5-dimensional torus with 262,144 nodes (fully populated systems). For the network performance model, we assume linear mapping of the MPI processes to the end nodes, an MPI node stack latency of 0.9 microseconds, a switch latency of 0.7 microseconds, a switch-to-switch link bandwidth of 5 GB/second, a node-to-switch link bandwidth of 5 GB/second, a node-to-switch link latency of 2.5 nanoseconds and a switch-to-switch link latency of 12 nanoseconds. For the network power model, we assume that the static power of a switch is 100 W (regardless of the radix value), the energy to TX/RX a bit through an electrical/optical link of 70 pJ. We show the design-space exploration results in Figure 7.9.

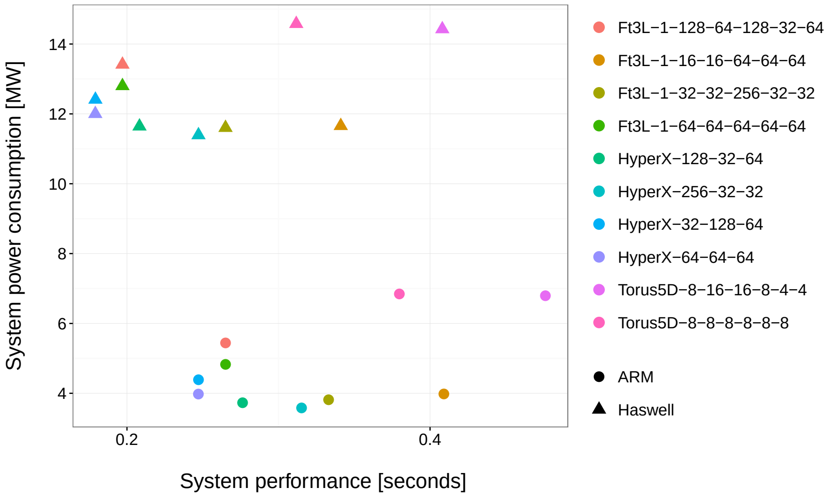


Figure 7.9 – Graph 500 design-space exploration.

In practice the system that we design has to conform with certain requirements. For example, there might be a constraint that the system should not consume more than 8 MW. If that is the only constraint, we can then select only the configurations under the 8 MW threshold. In our example this would exclude the Haswell-based systems and would expose the best choice as being a system with ARM-based compute nodes interconnected through a 2D HyperX topology with 64 nodes per switch and 64 switches in each dimension of the 2D topology. This system would achieve a performance of 0.24 seconds.

Conversely, for other systems, the more stringent requirements could be related to performance. Say we have systems where the constraint is that the application has to finish in less than 0.20 seconds. In this example the best choice corresponds to a system with Haswell-based compute nodes interconnected through a 2D HyperX topology with 64 nodes per switch and 64 switches in each dimension of the 2D topology. This system would achieve a performance of 0.18 seconds.

Even if the constraints are not clear at design time or might evolve, our methodology can generally provide a small set of Pareto-optimal configurations as shown in Figure 7.10.

In our example, the five points remaining after extracting the Pareto front expose a choice where the system designer can trade-off performance for power consumption, from the lowest-power configuration (e.g., ARM compute nodes interconnected via a 2D HyperX topology with 256 nodes per switch, 32 nodes in the X dimension and 32 nodes in the Y dimension of the 2D topology that would consume 3.58 MW for a performance of 0.31 seconds) to the highest-performing configuration (e.g., Haswell-based compute nodes interconnected through a 2D HyperX topology with 64 nodes per switch and 64 switches in each dimension of the 2D topology that would consume 12 MW for a performance of 0.18 seconds).

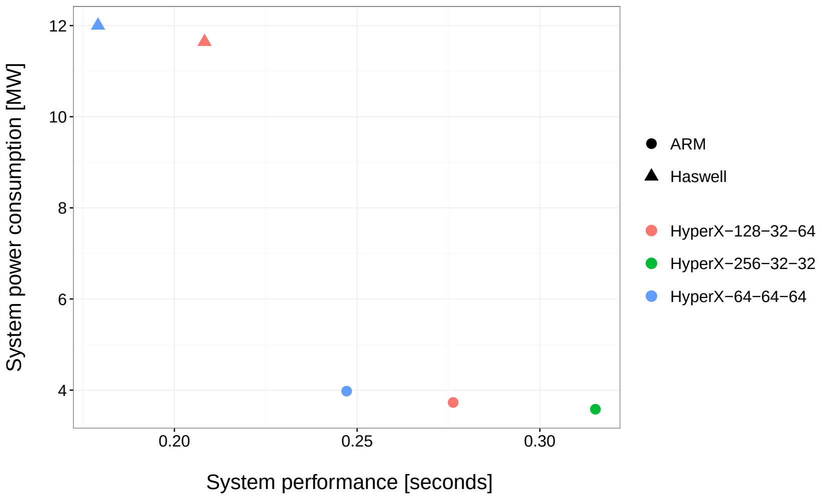


Figure 7.10 – Graph 500 design-space exploration - Pareto front.

Such studies are very useful at an early design-space exploration stage. By employing our methodology for full-system performance evaluation, the system designers can rapidly evaluate very large sets of hardware designs. From the vast amount of results, they can select, e.g., the Pareto optimal configurations, typically in a much smaller number than the initial set of configurations, and analyze them into further detail using more accurate performance evaluation tools such as simulators.

8 Conclusions and Future Work

In the upcoming decade it is expected that we reach the exascale computing age. Such systems will come, however, with stringent power and performance constraints. Thus, the system architects will need to explore a wide range of processor and network architectures to build an optimal system design. A holistic methodology for design-space exploration that covers not only multiple system components, but also multiple performance metrics, is required to provide the architects with a good understanding of the architectures, applications and their interactions. Such a methodology is useful for any organization that needs to run massively parallel software on very large computer systems.

8.1 Conclusions

The main part of this thesis was dedicated to studying if it is possible to perform early design-space exploration of large-scale systems by (1) decoupling the software characterization from performance and power modeling and extracting compute and communication properties inherent to applications, and (2) loading the platform-independent software properties into analytic processor and network models. We proposed such a methodology and validated it with measurements of real supercomputers. For the most scalable MPI implementation of Graph 500, a representative benchmark of graph analytics, we showed that the methodology is able to accurately preserve the ranking of system configurations based on their performance.

The research in this part of the thesis was conducted to answer multiple research questions. How to characterize the inherent, hardware-independent properties of sequential and parallel applications? How to use them with compute models and quickly evaluate the performance of many hardware designs? How to analytically model the network performance to decide which network topology best suits the communication requirements of a certain class of applications? What methodologies to use to efficiently evaluate the system-level performance of a wide range of hardware designs?

To address these questions, we first presented PISA, a framework for hardware- and ISA-agnostic workload characterization for sequential and parallel applications. We illustrated how our framework can be leveraged to extract application properties that impact the system performance, such as instruction mix, memory access patterns, branch behavior, instruction-level parallelism and inter-process communication behavior. When comparing PISA's results with measurements on actual processors, we found that PISA is capable of extracting the instruction mix of an application with high accuracy for POWER8 and with reasonable accuracy for x86 processors. Moreover, PISA's data reuse distribution estimates with good accuracy the L1 cache hit rate for the SPEC CPU2006 benchmarks when compared to both x86 and POWER8 processors. Furthermore, PISA generally provides an optimistic estimate for the branch misprediction rate and its branch-entropy-based predictions exhibit good linear correlation with actual measurements across applications.

We further presented the first analysis of how hardware- and ISA-agnostic software profiles, such as PISA's, can enable analytic performance modeling of processors. We showed that by loading PISA profiles into analytic approaches that model the processor events independently, we obtained an average time accuracy of 45% across the SPEC CPU2006 and Graph 500 benchmarks when compared with measurements on actual processors. We also obtained a good correlation factor of 0.84 across applications. When we load the software profiles into analytic approaches that take into account the interactions between processor events, the results showed a higher average time accuracy of 34% and a correlation factor of 0.97.

We also performed a detail analysis of a common characterization metric, the branch entropy, in order to analytically model the branch miss rate of a processor. We showed that there is a good linear correlation between the branch-entropy-based miss rates obtained from hardware-agnostic-based branching traces and measurements on current processors. By analyzing the correlation, we identified the first method to reverse engineer the history size of a hardware predictor. We also provided a first study about the limitations of branch entropy and proposed an approach to derive analytic models of the performance of branch predictors. Finally, we introduced the *max-outcome* branch predictor metric that is able to integrate not only the history size of a branch predictor, but also the size of the pattern table. When assuming infinite pattern table size, the *max-outcome* metric outperforms the branch miss rate estimates based on branch entropy on average with 17 percentage points.

Furthermore, we proposed a theoretical method for estimating the node injection bandwidth effectively sustained by a network. The method quantifies the impact of link contention bottlenecks on the node effective bandwidth. We derived analytic bandwidth models for the uniform pattern under fat-trees, tori, full-mesh and 2D HyperX topologies, for the shift pattern under full-mesh and fat-tree topologies and for the 2-dimensional nearest-neighbor pattern under fat-trees, full-mesh and 2D HyperX topologies. The validation results indicate that the proposed effective bandwidth models are not only accurate, but can also reliably be used to perform design-space exploration across network configurations not only across variations of configurations of the same network topology, but also across types of topologies. Indeed, we obtained high linear correlations of more than 0.89 between the model-based estimates and the simulation-based results for all patterns and network topologies under study. With these models we provide the community with means of fast design-space exploration across network topologies.

We further presented the first methodology that estimates the performance of large-scale systems using as input platform-independent software profiles loaded into analytic processor and network models. We evaluated our approach using two applications, Graph 500 and the NAS LU, which we ran on real systems with multiple sets of network configurations. For the Graph 500 benchmark, we obtained very good correlation results across different hardware systems. This indicates that the proposed methodology could reliably be used to (1) rank systems based on their performance, and (2) perform early and fast design-space exploration. For the NAS LU benchmark, we also obtained good correlation results when using PISA profiles with hardware models (no extrapolation). This is again an encouraging result for using our approach in the context of fast and early design-space exploration.

Another part of this thesis was dedicated to a methodology that accurately measures the time that parallel asynchronous applications spend in compute, communication and inter-process data dependencies. Such a methodology is an alternative to regular tracing tools that aim to quantify system bottlenecks. Our method can be used by researchers and engineers that build and optimize systems based on tracing information of applications executed on systems similar to the target system.

As out-of-the-box profiling tools do not differentiate between data transfer and data dependencies, with our profiling methodology, if an application is communication-bound, we can accurately quantify how much time is spent in data transfer and how much in inter-process data dependencies. This is relevant information for a system designer, because only the time spent in data transfer can be optimized by optimizing the interconnect fabric. Initial attempts of characterization using standard profiling exposed several limitations, mainly a high spatial and temporal overhead and a lack of support for data dependencies. Using our custom profiling approach, we addressed these issues and were able to target larger problem sizes and degrees of parallelism, while improving the accuracy of the characterization.

In the upcoming decades, supercomputers are expected to significantly increase their size, performance and energy efficiency, reaching the exascale computing age. Building such a system under very stringent power and performance constraints will be challenging. With this thesis we provided the scientific community with tools and methods for fast and early design-space exploration of large sets of hardware processor and network designs. For system design based on measurements of applications run on existing systems, we also provided a methodology to accurately characterize the time that asynchronous parallel applications spend not only in compute and communication, but also in inter-process data dependencies.

8.2 Future Work

With regard to *processor modeling*, a next step is to understand how to efficiently, through fast analytic models, use the spatio-temporal locality heat-maps to accurately quantify the cache miss rates. We also envision to extend PISA to extract application properties specifically relevant to GPU and accelerators performance modeling. For instance, the performance of a GPU is highly impacted by the host-device communication overhead. For this purpose, PISA can be extended with an analysis to quantify the amount of data communication between a host (CPU) and a GPU (accelerator) code. One possible approach would be to use PISA to analyze the parallel regions of an OpenMP implementation. Indeed,assuming that those parallel regions would actually be run on an accelerator, PISA could measure the amount of memory data used across different parallel regions and between the serial and parallel regions.

With regard to the *communication models*, we plan to extend the communication pattern-specific network models to other standard HPC patterns, such as bit reversal, bit complement and matrix transpose. This will enable to rapidly explore large sets of network topology design points and quantify the network performance for a comprehensive set of standard HPC patterns. We will also extend the network models to support mappings of multiple MPI processes on a single compute node and validate the models with measurements of real supercomputers. This aspect is important in order to model large-scale systems with efficient usage of the available compute nodes. Finally, we will analyze the accuracy of predicting the performance of OpenMP+MPI software implementations.

As alternative to the network bandwidth models proposed in this thesis, which are specific to a pair of communication pattern and network topology, we plan to also study how to efficiently use a more generic, but slower solution such as max-flow linear formulations. A possible approach of integrating such a method in our full-system methodology would be to estimate the node injection bandwidth for small network sizes using the max-flow-based formulations and apply linear regression (or other similar machine learning methods as those presented in [94]) to predict the bandwidth for larger networks.

With regard to the *full-system performance prediction methodology*, we will apply it to more MPI applications of uniform, shift and 2-dimensional nearest-neighbor communication patterns to further validate our proposed methodology. Moreover, to enforce the topological order of the compute and communication events in an MPI application, we envision to analyze the accuracy of a hardware-independent communication graph combined with analytic processor and network models and existing simulators, such as DIMEMAS.

Finally, we will extend our full-system analysis method with *cost models* for compute nodes, network links and switches to enable performance-power-cost trade-off analysis of large-scale systems. This is relevant as power and cost are key for infrastructure planning and budgeting. Adding the cost component would complete our methodology and provide system designers with means of selecting the most suitable hardware design points across the three metrics.

Appendices

Models of Average Link Latency / Average Number of Links

Uniform Pattern

Network topology	Average link latency
2L Fat-tree	$x_1 = (m_1 - 1) \cdot 2 \cdot l_0$ $x_2 = (m_2 - 1) \cdot m_1 \cdot (2 \cdot l_0 + 2 \cdot l_1)$ $\Longrightarrow \mathrm{l}_{\mathrm{link}} = \frac{x_1 + x_2}{m_1 \cdot m_2 - 1}$
3L Fat-tree	$x_1 = (m_1 - 1) \cdot 2 \cdot l_0$ $x_2 = (m_2 - 1) \cdot m_1 \cdot (2 \cdot l_0 + 2 \cdot l_1)$ $x_3 = m_1 \cdot m_2 \cdot (m_3 - 1) \cdot (2 \cdot l_0 + 2 \cdot l_1 + 2 \cdot l_2)$ $\Longrightarrow \mathrm{l}_{\mathrm{link}} = \frac{x_1 + x_2 + x_3}{m_1 \cdot m_2 \cdot m_3 - 1}$
2D HyperX	$x_1 = (p - 1) \cdot 2 \cdot l_0$ $x_{21} = p \cdot (d_1 - 1) \cdot (2 \cdot l_0 + l_1)$ $x_{22} = p \cdot (d_2 - 1) \cdot (2 \cdot l_0 + l_2)$ $x_{23} = p \cdot (d_1 - 1) \cdot (d_2 - 1) \cdot (2 \cdot l_0 + l_1 + l_2)$ $\Longrightarrow \mathrm{l}_{\mathrm{link}} = \frac{x_1 + x_{21} + x_{22} + x_{23}}{d_1 \cdot d_2 \cdot p - 1}$
1D Torus	$x_1 = 2 \cdot l_0 + \frac{\lceil \frac{d_1}{2} \rceil \cdot \lfloor \frac{d_1}{2} \rfloor \cdot l_1}{d_1} - \frac{2 \cdot l_0}{p \cdot d_1}$ $\Longrightarrow \mathrm{l}_{\mathrm{link}} = x_1$
2D Torus	$x_1 = 2 \cdot l_0 + \frac{\lceil \frac{d_1}{2} \rceil \cdot \lfloor \frac{d_1}{2} \rfloor \cdot l_1}{d_1} + \frac{\lceil \frac{d_2}{2} \rceil \cdot \lfloor \frac{d_2}{2} \rfloor \cdot l_2}{d_2} - \frac{2 \cdot l_0}{p \cdot d_1 \cdot d_2}$ $\Longrightarrow \mathrm{l}_{\mathrm{link}} = x_1$
3D Torus	$x_1 = 2 \cdot l_0 + \frac{\lceil \frac{d_1}{2} \rceil \cdot \lfloor \frac{d_1}{2} \rfloor \cdot l_1}{d_1} + \frac{\lceil \frac{d_2}{2} \rceil \cdot \lfloor \frac{d_2}{2} \rfloor \cdot l_2}{d_2} + \frac{\lceil \frac{d_3}{2} \rceil \cdot \lfloor \frac{d_3}{2} \rfloor \cdot l_3}{d_3} - \frac{2 \cdot l_0}{p \cdot d_1 \cdot d_2 \cdot d_3}$ $\Longrightarrow \mathrm{l}_{\mathrm{link}} = x_1$
Full-mesh	$x_1 = a \cdot p \cdot (p - 1) \cdot 2 \cdot l_0$ $x_2 = a \cdot p \cdot p \cdot (a - 1) \cdot (2 \cdot l_0 + l_1)$ $\Longrightarrow \mathrm{l}_{\mathrm{link}} = \frac{x_1 + x_2}{a \cdot p \cdot (a \cdot p - 1)}$

Table 1 – Average link latency models (uniform communication pattern).

Supercomputer Measurements - Graph 500 - Network Configurations

Topology	p	d_1	d_2	d_3	a	w_0	w_1	w_2	m_1	m_2	m_3	#Processes
Torus1D	1	4	–	–	–	–	–	–	–	–	–	4
2D Torus	1	2	2	–	–	–	–	–	–	–	–	4
	1	2	4	–	–	–	–	–	–	–	–	8
	1	4	4	–	–	–	–	–	–	–	–	16
3D Torus	1	2	2	2	–	–	–	–	–	–	–	8
	1	2	2	4	–	–	–	–	–	–	–	16
	1	2	4	4	–	–	–	–	–	–	–	32
	1	4	4	4	–	–	–	–	–	–	–	64
2D HyperX	1	2	2	–	–	–	–	–	–	–	–	4
	1	4	2	–	–	–	–	–	–	–	–	8
	2	2	2	–	–	–	–	–	–	–	–	8
	1	4	4	–	–	–	–	–	–	–	–	16
	1	8	2	–	–	–	–	–	–	–	–	16
	2	4	2	–	–	–	–	–	–	–	–	16
	4	2	2	–	–	–	–	–	–	–	–	16
	1	8	4	–	–	–	–	–	–	–	–	32
	2	4	4	–	–	–	–	–	–	–	–	32
	2	8	2	–	–	–	–	–	–	–	–	32
	4	4	2	–	–	–	–	–	–	–	–	32
	2	8	4	–	–	–	–	–	–	–	–	64
	4	4	4	–	–	–	–	–	–	–	–	64
	4	8	2	–	–	–	–	–	–	–	–	64
Full-mesh	1	–	–	–	4	–	–	–	–	–	–	4
	1	–	–	–	8	–	–	–	–	–	–	8
	2	–	–	–	4	–	–	–	–	–	–	8
	1	–	–	–	16	–	–	–	–	–	–	16
	2	–	–	–	8	–	–	–	–	–	–	16
	4	–	–	–	4	–	–	–	–	–	–	16
	2	–	–	–	16	–	–	–	–	–	–	32
	4	–	–	–	8	–	–	–	–	–	–	32
	4	–	–	–	16	–	–	–	–	–	–	64
2L Fat-tree	–	–	–	–	–	1	4	–	1	4	–	4
	–	–	–	–	–	1	4	–	2	4	–	8
	–	–	–	–	–	1	4	–	4	4	–	16
	–	–	–	–	–	1	4	–	8	4	–	32
	–	–	–	–	–	1	4	–	16	4	–	64
3L Fat-tree	–	–	–	–	–	1	4	2	1	4	2	8
	–	–	–	–	–	1	4	2	2	4	2	16
	–	–	–	–	–	1	4	2	4	4	2	32
	–	–	–	–	–	1	4	2	8	4	2	64

Table 2 – Network topology configurations (Graph 500).

Models of Average Link Latency / Average Number of Links
2-Dimensional Nearest-Neighbor Pattern (1)

Mapping description	**Average link latency**
$d_1^1 = D_1 \wedge d_2^1 \mid D_2$	$x_1 = 2 \cdot d_1^1 \cdot (2 \cdot l_0 + l_1)$ $x_2 = (4 \cdot d_1^1 \cdot d_2^1 - 2 \cdot d_1^1) \cdot 2 \cdot l_0$ $\Longrightarrow \mathrm{l}_{\mathrm{link}} = \frac{x_1 + x_2}{4 \cdot d_1^1 \cdot d_2^1}$
$d_1^1 \mid D_1 \wedge d_2^1 = D_2$	$x_1 = 2 \cdot d_2^1 \cdot (2 \cdot l_0 + l_1)$ $x_2 = (4 \cdot d_1^1 \cdot d_2^1 - 2 \cdot d_2^1) \cdot 2 \cdot l_0$ $\Longrightarrow \mathrm{l}_{\mathrm{link}} = \frac{x_1 + x_2}{4 \cdot d_1^1 \cdot d_2^1}$
$d_2^1 \mid D_2 \wedge d_1^1 \mid D_1$	$x_1 = (2 \cdot d_2^1 + 2 \cdot d_1^1) \cdot (2 \cdot l_0 + l_1)$ $x_2 = (4 \cdot d_1^1 \cdot d_2^1 - 2 \cdot d_2^1 - 2 \cdot d_1^1) \cdot 2 \cdot l_0$ $\Longrightarrow \mathrm{l}_{\mathrm{link}} = \frac{x_1 + x_2}{4 \cdot d_1^1 \cdot d_2^1}$

Table 3 – Average link latency models (nearest-neighbor, full-mesh).

Mapping description	**Average link latency**
$d_1^1 = D_1 \wedge d_2^1 \mid D_2$	$x_1 = 2 \cdot d_1^1 \cdot (2 \cdot l_0 + 2 \cdot l_1)$ $x_2 = (4 \cdot d_1^1 \cdot d_2^1 - 2 \cdot d_1^1) \cdot 2 \cdot l_0$ $\Longrightarrow \mathrm{l}_{\mathrm{link}} = \frac{x_1 + x_2}{4 \cdot d_1^1 \cdot d_2^1}$
$d_2^1 = D_2 \wedge d_1^1 \mid D_1$	$x_1 = 2 \cdot d_2^1 \cdot (2 \cdot l_0 + 2 \cdot l_1)$ $x_2 = (4 \cdot d_1^1 \cdot d_2^1 - 2 \cdot d_2^1) \cdot 2 \cdot l_0$ $\Longrightarrow \mathrm{l}_{\mathrm{link}} = \frac{x_1 + x_2}{4 \cdot d_1^1 \cdot d_2^1}$
$d_2^1 \mid D_2 \wedge d_1^1 \mid D_1$	$x_1 = (2 \cdot d_2^1 + 2 \cdot d_1^1) \cdot (2 \cdot l_0 + 2 \cdot l_1)$ $x_2 = (4 \cdot d_1^1 \cdot d_2^1 - 2 \cdot d_2^1 - 2 \cdot d_1^1) \cdot 2 \cdot l_0$ $\Longrightarrow \mathrm{l}_{\mathrm{link}} = \frac{x_1 + x_2}{4 \cdot d_1^1 \cdot d_2^1}$

Table 4 – Average link latency models (nearest-neighbor, 2L fat-tree).

Models of Average Link Latency / Average Number of Links
2-Dimensional Nearest-Neighbor Pattern (2)

Mapping description	**Average link latency**
$d_1^2 = d_1^1 = D_1 \wedge d_2^2 \vert d_2^1 \vert D_2$	$x_1 = 2 \cdot d_2^2 \cdot (2 \cdot l_0 + 2 \cdot l_1 + 2 \cdot l_2)$ $x_2 = (4 \cdot d_1^1 \cdot d_2^1 - 4 \cdot d_1^2 \cdot d_2^2) \cdot (2 \cdot l_0 + 2 \cdot l_1)$ $x_3 = (4 \cdot d_1^2 \cdot d_2^2 - 2 \cdot d_1^2) \cdot 2 \cdot l_0$ $\Longrightarrow \mathrm{l_{link}} = \frac{x_1 + x_2 + x_3}{4 \cdot d_1^1 \cdot d_2^1}$
$d_1^2 \vert d_1^1 = D_1 \wedge d_2^2 = d_2^1 \vert D_2$ $d_1^2 \vert d_1^1 = D_1 \wedge d_2^2 \vert d_2^1 \vert D_2$	$x_1 = 2 \cdot d_2^2 \cdot (2 \cdot l_0 + 2 \cdot l_1 + 2 \cdot l_2)$ $x_2 = (4 \cdot d_1^1 \cdot d_2^1 - 4 \cdot d_1^2 \cdot d_2^2 + 2 \cdot d_1^2 + 2 \cdot d_2^2 - 2 \cdot d_1^1) \cdot (2 \cdot l_0 + 2 \cdot l_1)$ $x_3 = (4 \cdot d_1^2 \cdot d_2^2 - 2 \cdot d_1^2 - 2 \cdot d_2^2) \cdot 2 \cdot l_0$ $\Longrightarrow \mathrm{l_{link}} = \frac{x_1 + x_2 + x_3}{4 \cdot d_1^1 \cdot d_2^1}$
$d_1^2 \vert d_1^1 \vert D_1 \wedge d_2^2 = d_2^1 = D_2$	$x_1 = 2 \cdot d_2^1 \cdot (2 \cdot l_0 + 2 \cdot l_1 + 2 \cdot l_2)$ $x_2 = (4 \cdot d_1^1 \cdot d_2^1 - 4 \cdot d_1^2 \cdot d_2^2) \cdot (2 \cdot l_0 + 2 \cdot l_1)$ $x_3 = (4 \cdot d_1^2 \cdot d_2^2 - 2 \cdot d_2^2) \cdot 2 \cdot l_0$ $\Longrightarrow \mathrm{l_{link}} = \frac{x_1 + x_2 + x_3}{4 \cdot d_1^1 \cdot d_2^1}$
$d_1^2 = d_1^1 \vert D_1 \wedge d_2^2 \vert d_2^1 = D_2$ $d_1^2 \vert d_1^1 \vert D_1 \wedge d_2^2 \vert d_2^1 = D_2$	$x_1 = 2 \cdot d_2^1 \cdot (2 \cdot l_0 + 2 \cdot l_1 + 2 \cdot l_2)$ $x_2 = (4 \cdot d_1^1 \cdot d_2^1 - 4 \cdot d_1^2 \cdot d_2^2 + 2 \cdot d_1^2 + 2 \cdot d_2^2 - 2 \cdot d_2^1) \cdot (2 \cdot l_0 + 2 \cdot l_1)$ $x_3 = (4 \cdot d_1^2 \cdot d_2^2 - 2 \cdot d_1^2 - 2 \cdot d_2^2) \cdot 2 \cdot l_0$ $\Longrightarrow \mathrm{l_{link}} = \frac{x_1 + x_2 + x_3}{4 \cdot d_1^1 \cdot d_2^1}$
$d_1^2 = d_1^1 \vert D_1 \wedge d_2^2 \vert d_2^1 \vert D_2$ $d_1^2 \vert d_1^1 \vert D_1 \wedge d_2^2 = d_2^1 \vert D_2$ $d_1^2 \vert d_1^1 \vert D_1 \wedge d_2^2 \vert d_2^1 \vert D_2$	$x_1 = (2 \cdot d_1^1 + 2 \cdot d_2^1) \cdot (2 \cdot l_0 + 2 \cdot l_1 + 2 \cdot l_2)$ $x_2 = (4 \cdot d_1^1 \cdot d_2^1 - 4 \cdot d_1^2 \cdot d_2^2 + 2 \cdot d_1^2 + 2 \cdot d_2^2 - 2 \cdot d_1^1 - 2 \cdot d_2^1) \cdot (2 \cdot l_0 + 2 \cdot l_1)$ $x_3 = (4 \cdot d_1^2 \cdot d_2^2 - 2 \cdot d_1^2 - 2 \cdot d_2^2) \cdot 2 \cdot l_0$ $\Longrightarrow \mathrm{l_{link}} = \frac{x_1 + x_2 + x_3}{4 \cdot d_1^1 \cdot d_2^1}$

Table 5 – Average link latency models (nearest-neighbor, 3L fat-tree).

Models of Average Link Latency / Average Number of Links

2-Dimensional Nearest-Neighbor Pattern (3)

Mapping description	Average link latency
$m = 1 \wedge k = 1$	$x_1 = 4 \cdot l_0 \cdot p \cdot m$ $x_2 = (2 \cdot l_0 + l_2) \cdot 2 \cdot p \cdot m$ $\Longrightarrow \mathrm{l}_{\mathrm{link}} = \frac{x_1+x_2}{2 \cdot p \cdot m + 2 \cdot p \cdot m}$
$m = 1 \wedge k = 2$	$x_1 = 4 \cdot l_0 \cdot k \cdot p \cdot m$ $x_2 = (2 \cdot l_0 + l_1) \cdot 2 \cdot p \cdot m$ $x_3 = (2 \cdot l_0 + l_1 + l_2) \cdot 2 \cdot p \cdot m$ $\Longrightarrow \mathrm{l}_{\mathrm{link}} = \frac{x_1+x_2+x_3}{2 \cdot k \cdot p \cdot m + 2 \cdot p \cdot m + 2 \cdot p \cdot m}$
$m = 1 \wedge k > 2$	$x_1 = 4 \cdot l_0 \cdot k \cdot p \cdot m$ $x_2 = (2 \cdot l_0 + l_1) \cdot 2 \cdot p \cdot m$ $x_3 = (2 \cdot l_0 + l_1 + l_2) \cdot 2 \cdot p \cdot m$ $x_4 = (2 \cdot l_0 + l_1) \cdot (k-2) \cdot 2 \cdot p \cdot m$ $\Longrightarrow \mathrm{l}_{\mathrm{link}} = \frac{x_1+x_2+x_3+x_4}{2 \cdot k \cdot p \cdot m + 2 \cdot p \cdot m + 2 \cdot p \cdot m + (k-2) \cdot 2 \cdot p \cdot m}$
$m >= 2 \wedge k = 1$	$x_1 = 2 \cdot l_0 \cdot (2 \cdot p \cdot m - 2 \cdot m)$ $x_2 = (2 \cdot l_0 + l_1) \cdot 2 \cdot m$ $x_3 = (2 \cdot l_0 + l_2) \cdot 2 \cdot p \cdot m$ $\Longrightarrow \mathrm{l}_{\mathrm{link}} = \frac{x_1+x_2+x_3}{2 \cdot p \cdot m - 2 \cdot m + 2 \cdot m + 2 \cdot p \cdot m}$
$m >= 2 \wedge k = 2$	$x_1 = 2 \cdot l_0 \cdot k \cdot (2 \cdot p \cdot m - 2 \cdot m)$ $x_2 = (2 \cdot l_0 + l_1) \cdot 2 \cdot k \cdot m$ $x_3 = (2 \cdot l_0 + l_1) \cdot 2 \cdot p \cdot m$ $x_4 = (2 \cdot l_0 + l_1 + l_2) \cdot 2 \cdot p \cdot m$ $\Longrightarrow \mathrm{l}_{\mathrm{link}} = \frac{x_1+x_2+x_3+x_4}{2 \cdot k \cdot p \cdot m - 2 \cdot k \cdot m + 2 \cdot k \cdot m + 2 \cdot p \cdot m + 2 \cdot p \cdot m}$
$m >= 2 \wedge k > 2$	$x_1 = 2 \cdot l_0 \cdot k \cdot (2 \cdot p \cdot m - 2 \cdot m)$ $x_2 = (2 \cdot l_0 + l_1) \cdot k \cdot 2 \cdot m$ $x_3 = (2 \cdot l_0 + l_1) \cdot 2 \cdot p \cdot m$ $x_4 = (2 \cdot l_0 + l_1 + l_2) \cdot 2 \cdot p \cdot m$ $x_5 = (2 \cdot l_0 + l_1) \cdot (k-2) \cdot 2 \cdot p \cdot m$ $\Longrightarrow \mathrm{l}_{\mathrm{link}} = \frac{x_1+x_2+x_3+x_4+x_5}{2 \cdot k \cdot p \cdot m - 2 \cdot k \cdot m + 2 \cdot k \cdot m + 2 \cdot p \cdot m + 2 \cdot p \cdot m + (k-2) \cdot 2 \cdot p \cdot m}$

Table 6 – Average link latency models (nearest-neighbor, 2D HyperX).

Supercomputer Measurements - NAS LU - Network Configurations

Topology	p	d_1	d_2	d_3	a	w_0	w_1	w_2	m_1	m_2	m_3	#Processes
2D HyperX	4	2	2	–	–	–	–	–	–	–	–	16
	2	4	2	–	–	–	–	–	–	–	–	16
	1	4	4	–	–	–	–	–	–	–	–	16
	1	8	2	–	–	–	–	–	–	–	–	16
	1	5	5	–	–	–	–	–	–	–	–	25
	1	12	3	–	–	–	–	–	–	–	–	36
	3	4	3	–	–	–	–	–	–	–	–	36
	3	6	2	–	–	–	–	–	–	–	–	36
	2	6	3	–	–	–	–	–	–	–	–	36
	1	6	6	–	–	–	–	–	–	–	–	36
	2	9	2	–	–	–	–	–	–	–	–	36
	4	4	4	–	–	–	–	–	–	–	–	64
	2	8	4	–	–	–	–	–	–	–	–	64
Full-mesh	2	–	–	–	8	–	–	–	–	–	–	16
	1	–	–	–	16	–	–	–	–	–	–	16
	4	–	–	–	16	–	–	–	–	–	–	64
2L Fat-tree	–	–	–	–	–	1	4	–	4	4	–	16
	–	–	–	–	–	1	4	–	16	4	–	64
3L Fat-tree	–	–	–	–	–	1	4	2	2	4	2	16
	–	–	–	–	–	1	4	2	8	4	2	64

Table 7 – Network topology configurations (nearest-neighbor pattern).

Author's Publications and Patents

Journal Peer-Reviewed Articles

A. Anghel, L. Vasilescu, G. Mariani, R. Jongerius, G. Dittmann, "An Instrumentation Approach for Hardware-Agnostic Software Characterization," in *International Journal on Parallel Programming (IJPP'16)* 44: 924, Springer, 2016, DOI:10.1007/s10766-016-0410-0.

G. Mariani, **A. Anghel**, R. Jongerius, G. Dittmann, "Scaling Application Properties to Exascale," in *International Journal on Parallel Programming (IJPP'16)*, 44: 975, Springer, 2016, DOI:10.1007/s10766-016-0412-y.

G. Mariani, **A. Anghel**, R. Jongerius, G. Dittmann, "Classification of Thread Profiles for Scaling Application Behavior," in *Journal of Parallel Computing*, Elsevier, 2017.

R. Jongerius, **A. Anghel**, G. Mariani, G. Dittmann, E. Vermij, H. Corporaal, "Analytic Multi-Core Processor Model for Fast Design-Space Exploration,", in *IEEE Transactions on Computers (TC)*, IEEE 2017 (under review).

Conference Peer-Reviewed Articles

A. Anghel, L. Vasilescu, G. Mariani, R. Jongerius, G. Dittmann, "An Instrumentation Approach for Hardware-Agnostic Software Characterization," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*, ser. CF'15. New York, NY, USA: ACM, 2015, pp. 3:1–3:8, DOI: http://dx.doi.org/10.1145/2742854.2742859.

G. Mariani, **A. Anghel**, R. Jongerius, G. Dittmann, "Scaling Application Properties to Exascale," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*, ser. CF'15. New York, USA: ACM, 2015, pp. 31:1–31:8, DOI: http://dx.doi.org/10.1145/2742854.2742860.

A. Anghel, G. Mariani, R. Jongerius, G. Dittmann, "Analytic Performance Modeling of Fat-Tree Topologies For Fast Design-Space Exploration", in *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'18)*, IEEE 2018 (under review).

R. Jongerius, G. Mariani, **A. Anghel**, E. Vermij, G. Dittmann, H. Corporaal, "Analytic processor model for fast design-space exploration,” in *International IEEE Conference on Computer Design (ICCD’15)*, New York, USA, IEEE, 2015, DOI: 10.1109/ICCD.2015.7357136.

S. Poddar, R. Jongerius, F. Leandro, G. Mariani, G. Dittmann, **A. Anghel**, H. Corporaal, “MeSAP: A fast analytic power model for DRAM memories,” in *Design, Automation and Test in Europe Conference*, pp. 49–54, IEEE, 2017, DOI: 10.23919/DATE.2017.7926957.

A. Anghel, G. Rodriguez, B. Prisacari, C. Minkenberg, G. Dittmann, “Quantifying Communication in Graph Analytics,"" in *Proceedings of High Performance Computing - 30th International Conference, ISC High Performance 2015*, pp. 472–487, 2015, Lecture Notes in Computer Science, vol 9137, Springer, Cham.

P. Fuentes, E. Vallejo, J.L. Bosque, R. Beivide, **A. Anghel**, G. Rodriguez, M. Gusat, C. Minkenberg, “Synthetic Traffic Model of the Graph500 Communications,” in *Proceedings of the 16th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, 2016, Lecture Notes in Computer Science, vol 10048, Springer, Cham.

G. Kathareios, **A. Anghel**, A. Mate, R. Clauberg, Mitch Gusat, “Catch It If You Can: Real-Time Network Anomaly Detection With Low False Alarm Rates“, in *Proceedings of the 16th IEEE International Conference On Machine Learning And Applications (ICMLA’17)*, IEEE, 2017.

A. Anghel, R. Jongerius, G. Dittmann, J. Weiss, and R. P. Luijten, “Holistic power analysis of implementation alternatives for a very large scale synthesis array with phased array stations,” in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2014*, pp. 5397–5401, IEEE, 2014, DOI: 10.1109/ICASSP.2014.6854634.

M. L. Schmatz, R. Jongerius, G. Dittmann, **A. Anghel**, T. Engbersen, J. van Lunteren, and P. Buchmann, “Scalable, efficient ASICs for the square kilometre array: From A/D conversion to central correlation,” in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2014*, pp. 7505–7509, IEEE, 2014, DOI: 10.1109/ICASSP.2014.6855059.

A. Anghel, G. Rodriguez, and B. Prisacari, “The importance and characteristics of communication in high-performance data analytics,” in *Proceedings of the IEEE International Symposium on Workload Characterization, IISWC 2014*, IEEE, 2014, DOI: 10.1109/IISWC.2014.6983044.

D. Crisan, **A. Anghel**, R. Birke, C. Minkenberg, and M. Gusat, “Short and fat: TCP performance in CEE datacenter networks,” in *Proceedings of the IEEE 19th Annual Symposium on High Performance Interconnects, HOTI 2011*, pp. 43–50, IEEE, 2011, DOI: 10.1109/HOTI.2011.16.

Workshop Peer-Reviewed Articles

A. Anghel, R. Birke, and M. Gusat, “Scalable high resolution traffic heatmaps: Coherent queue visualization for datacenters,” in *Proceedings of the 6th International Workshop on Traffic*

Monitoring and Analysis TMA 2014, pp. 26–37, 2014, Lecture Notes in Computer Science, vol 8406, Springer, Berlin, Heidelberg.

A. Anghel, G. Dittmann, R. Jongerius, and R. Luijten, "Spatio-temporal locality characterization, in *46th IEEE/ACM International Symposium on Microarchitecture (MICRO-46) Workshops: 1st Workshop on Near-Data Processing,* 2013." [Online].
Available: http://www.cs.utah.edu/wondp/Anghel_Locality.pdf

A. Anghel, R. Birke, D. Crisan, and M. Gusat, "Cross-layer flow and congestion control for datacenter networks," in *23rd International Teletraffic Congress (ITC 23) Workshops: 3rd Workshop on Data Center Converged and Virtual Ethernet Switching, DC-CAVES, 2011, San Francisco, CA, USA, September 9, 2011.*

Patents

A. Anghel, G. Dittmann, P. Altevogt, C. Lichtenau, T. Pflueger, IBM Corporation, "Cognitive Selection of Variable Memory-Access Widths", Patent Office CH, Patent Number CH820160129, 2017.

A. Anghel, G. Dittmann, P. Altevogt, C. Lichtenau, T. Pflueger, IBM Corporation, "Cognitive Load-Aware Branch Prediction", Patent Office CH, Patent Number CH820160099, 2017.

A. Anghel, B. Prisacari, IBM Corporation, Large Scale Distributed Training of Machine Learning Models Without Loss of Accuracy, Patent Number CH920160046US1, 2016.

A. Anghel, G. Kathareios, M. Gusat, IBM Corporation, Monitoring Queues at Switches of a Network from an External Entity, Patent Office US, Patent Number CH920150073US1, 2015.

A. Anghel, B. Prisacari, G. Rodriguez, IBM Corporation, Datacenter Scheduling of Applications Using Machine Learning Techniques, Patent Office US, Patent Number CH920140079US1, 2015.

A. Anghel, C. Basso, R. Birke, D. Crisan, M. Gusat, K. Kamble, C. Minkenberg, IBM Corporation, Quantized Congestion Notification (QCN) Extension to Explicit Congestion Notification (ECN) for Transport-based End-to-End Congestion Notification, Patent Office US, Patent Number 20150188820, Application number 14/145683, Publication date 2015/07.

A. Anghel, R. Birke, C. DeCusatis, M. Gusat, C. Minkenberg, IBM Corporation, Coherent Load Monitoring of Physical and Virtual Networks With Synchronous Status Acquisition, Patent Office US, Patent Number US20140269403 A1, Application Number US 13/834,679, Publication date 2014/09.

Bibliography

[1] Apache Giraph. http://giraph.apache.org/.

[2] Barcelona Supercomputing Center MareNostrum supercomputer. https://www.bsc.es/innovation-and-services/supercomputers-and-facilities/marenostrum.

[3] Extrae. Barcelona Supercomputer Center. https://tools.bsc.es/extrae.

[4] Graph 500 benchmark. http://www.graph500.org/.

[5] Intel MPI Benchmarks, User Guide and Methodology Description (version 3.2.4). https://www.lrz.de/services/compute/courses/x_lecturenotes/mic_workshop_ostrava/IMB_Users_Guide.pdf.

[6] International Technology Roadmap for Semiconductors (ITRS). http://www.itrs2.net/itrs-reports.html (accessed November 2013).

[7] Magnus technical specifications. https://www.pawsey.org.au/our-systems/magnus-technical-specifications (accessed September 2016).

[8] OMNEST, High-Performance Simulation for All Kinds of Networks. https://omnest.com/.

[9] OMNeT++, Discrete Event Simulator. https://omnetpp.org/.

[10] Standard Performance Evaluation Corporation, SPEC MPI2007. https://www.spec.org/mpi/.

[11] Top 500 list. http://www.top500.org.

[12] Top 500 list. http://www.top500.org/list/2014/11/ (November 2014).

[13] A2 Processor – User's Manual for Blue Gene/Q (version 1.3). https://computing.llnl.gov/tutorials/bgq/A2UserManual.pdf, 2012.

[14] ADIGA, N. R., BLUMRICH, M. A., CHEN, D., COTEUS, P., GARA, A., GIAMPAPA, M. E., HEIDELBERGER, P., SINGH, S., STEINMACHER-BUROW, B. D., TAKKEN, T., TSAO, M., AND

VRANAS, P. Blue Gene/L torus interconnection network. *IBM J. Res. Dev. 49*, 2 (Mar. 2005), 265–276.

[15] AGARWAL, V., PETRINI, F., PASETTO, D., AND BADER, D. A. Scalable graph exploration on multicore processors. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (Washington, DC, USA, 2010), SC '10, IEEE Computer Society, pp. 1–11.

[16] AHN, J. H., BINKERT, N., DAVIS, A., MCLAREN, M., AND SCHREIBER, R. S. HyperX: Topology, routing, and packaging of efficient large-scale networks. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (New York, NY, USA, 2009), SC '09, ACM, pp. 41:1–41:11.

[17] AILAMAKI, A., DEWITT, D. J., HILL, M. D., AND WOOD, D. A. DBMSs on a Modern Processor: Where Does Time Go? In *Proceedings of the 25th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1999), VLDB '99, Morgan Kaufmann Publishers Inc., pp. 266–277.

[18] AL-FARES, M., LOUKISSAS, A., AND VAHDAT, A. A scalable, commodity data center network architecture. *SIGCOMM Comput. Commun. Rev. 38*, 4 (Aug. 2008), 63–74.

[19] ALVERSON, B., FROESE, E., KAPLAN, L., AND INC.), D. R. C. Cray XC series network. http://www.cray.com/sites/default/files/resources/CrayXCNetwork.pdf (accessed September 2016).

[20] AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference* (New York, NY, USA, 1967), AFIPS '67 (Spring), ACM, pp. 483–485.

[21] ANGHEL, A., RODRIGUEZ, G., PRISACARI, B., MINKENBERG, C., AND DITTMANN, G. Quantifying communication in graph analytics. In *High Performance Computing*, J. M. Kunkel and T. Ludwig, Eds., vol. 9137 of *Lecture Notes in Computer Science*. Springer International Publishing, 2015, pp. 472–487.

[22] ANGHEL, A., VASILESCU, L. M., JONGERIUS, R., DITTMANN, G., AND MARIANI, G. An instrumentation approach for hardware-agnostic software characterization. In *Proceedings of the 12th ACM International Conference on Computing Frontiers* (New York, NY, USA, 2015), CF '15, ACM, pp. 3:1–3:8.

[23] ANGHEL, A., VASILESCU, L. M., MARIANI, G., JONGERIUS, R., AND DITTMANN, G. An instrumentation approach for hardware-agnostic software characterization. *International Journal of Parallel Programming 44*, 5 (2016), 924–948.

[24] APPLEGATE, D., AND COHEN, E. Making intra-domain routing robust to changing and uncertain traffic demands: Understanding fundamental tradeoffs. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (New York, NY, USA, 2003), SIGCOMM '03, ACM, pp. 313–324.

[25] ARGOLLO, E., FALCÓN, A., FARABOSCHI, P., MONCHIERO, M., AND ORTEGA, D. COTSon: Infrastructure for full system simulation. *SIGOPS Oper. Syst. Rev. 43*, 1 (Jan. 2009), 52–61.

[26] ASHBY, S., BECKMAN, P., CHEN, J., COLELLA, P., COLLINS, B., CRAWFORD, D., DONGARRA, J., KOTHE, D., LUSK, R., MESSINA, P., AND OTHERS. The opportunities and challenges of exascale computing. *Summary report of the advanced scientific computing advisory committee (ASCAC) subcommittee at the US Department of Energy Office of Science* (2010).

[27] AUSTIN, T., LARSON, E., AND ERNST, D. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer 35*, 2 (Feb. 2002), 59–67.

[28] AUSTIN, T. M., AND SOHI, G. S. Dynamic dependency analysis of ordinary programs. *SIGARCH Comput. Archit. News 20*, 2 (Apr. 1992), 342–351.

[29] AZAR, Y., COHEN, E., FIAT, A., KAPLAN, H., AND RACKE, H. Optimal oblivious routing in polynomial time. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 2003), STOC '03, ACM, pp. 383–388.

[30] BADAWY, A.-H., AGGARWAL, A., YEUNG, D., AND TSENG, C.-W. The efficacy of software prefetching and locality optimizations on future memory systems. *Journal of Instruction-Level Parallelism 6*, 7 (2004).

[31] BADER, D., RIEDY, J., AND MEYERHENKE, H. Applications and challenges in large-scale graph analysis. *HPC Graph Analytics Workshop* (2013).

[32] BADIA, R. M., LABARTA, J., GIMENEZ, J., AND ESCALE, F. DIMEMAS: Predicting MPI applications behavior in grid environments. In *Workshop on Grid Applications and Programming Tools (GGF8)* (2003), vol. 86, pp. 52–62.

[33] BAILEY, D., HARRIS, T., SAPHIR, W., VAN DER WIJNGAART, R., WOO, A., AND YARROW, M. The NAS Parallel Benchmarks 2.0. *The International Journal of Supercomputer Applications* (1995).

[34] BAUDRY, A., LACASSE, R., ESCOFFIER, R., WEBBER, J., GREENBERG, J., PLATT, L., TREACY, R., SAEZ, A. F., CAIS, P., COMORETTO, G., QUERTIER, B., OKUMURA, S. K., KAMAZAKI, T., CHIKADA, Y., WATANABE, M., OKUDA, T., KURONO, Y., AND IGUCHI, S. Performance highlights of the ALMA correlators, in Proc. SPIE 8452, Millimeter, Submillimeter, and Far-Infrared Detectors and Instrumentation for Astronomy VI, 2012.

[35] BECKMANN, N., EASTEP, J., GRUENWALD, C., KURIAN, G., KASTURE, H., MILLER, J. E., CELIO, C., AND AGARWAL, A. Graphite: A Distributed Parallel Simulator for Multicores. Tech. rep., MIT, Nov. 2009.

[36] BEYLS, K., AND D'HOLLANDER, E. Reuse distance as a metric for cache behavior. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems* (2001), pp. 617–662.

[37] BHUYAN, L. N., AND AGRAWAL, D. P. Generalized hypercube and hyperbus structures for a computer network. *IEEE Trans. Comput. 33*, 4 (Apr. 1984), 323–333.

[38] BIENIA, C., KUMAR, S., SINGH, J. P., AND LI, K. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (New York, NY, USA, 2008), PACT '08, ACM, pp. 72–81.

[39] BINKERT, N., BECKMANN, B., BLACK, G., REINHARDT, S. K., SAIDI, A., BASU, A., HESTNESS, J., HOWER, D. R., KRISHNA, T., SARDASHTI, S., SEN, R., SEWELL, K., SHOAIB, M., VAISH, N., HILL, M. D., AND WOOD, D. A. The Gem5 simulator. *SIGARCH Comput. Archit. News 39*, 2 (Aug. 2011), 1–7.

[40] BIRAN, A. Characterization of the Cray Aries network (NERCS advanced technology group). https://www.nersc.gov/assets/pubs_presos/NUG2014Aries.pdf (accessed September 2016), 2014.

[41] BORKAR, S., AND CHIEN, A. A. The future of microprocessors. *Commun. ACM 54*, 5 (May 2011), 67–77.

[42] BROEKEMA, P. C., BOONSTRA, A.-J., CABEZAS, V. C., ENGBERSEN, T., HOLTIES, H., JELITTO, J., LUIJTEN, R. P., MAAT, P., VAN NIEUWPOORT, R. V., NIJBOER, R., ROMEIN, J. W., AND OFFREIN, B. J. DOME: Towards the ASTRON and IBM Center for Exascale Technology. In *Proceedings of the 2012 Workshop on High-Performance Computing for Astronomy Date* (New York, NY, USA, 2012), ACM.

[43] CABEZAS, V. A tool for analysis and visualization of application properties. Tech. Rep. RZ3834, IBM, 2012.

[44] CALZAROSSA, M., AND SERAZZI, G. Workload Characterization: a survey. *Proceedings of the IEEE 8*, 81 (1993), 1136–1150.

[45] CARLSON, T. E., HEIRMAN, W., EYERMAN, S., HUR, I., AND EECKHOUT, L. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization (TACO)* (2014).

[46] CHAKRABARTI, D., ZHAN, Y., AND FALOUTSOS, C. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining.*

[47] CHANG, P.-Y., HAO, E., YEH, T.-Y., AND PATT, Y. Branch classification: A new mechanism for improving branch predictor performance. *International Journal of Parallel Programming 24*, 2 (1996), 133–158.

[48] CHANGWOO, P., KYUNG-WOO, L., HYE-KYUNG, H., AND GYUNGHO, L. Reference distance as a metric for data locality. In *Proceedings of HPC-ASIA97* (1997), pp. 151–156.

[49] CHECCONI, F., AND PETRINI, F. Massive data analytics: The Graph 500 on IBM Blue Gene/Q. *IBM Journal of Research and Development 57*, 1/2 (2013), 10.

[50] CHEN, D., EISLEY, N. A., HEIDELBERGER, P., SENGER, R. M., SUGAWARA, Y., KUMAR, S., SALAPURA, V., SATTERFIELD, D., STEINMACHER-BUROW, B., AND PARKER, J. The IBM Blue Gene/Q interconnection fabric. *IEEE Micro 32* (2012), 32–43.

[51] CHEN, D., EISLEY, N. A., HEIDELBERGER, P., SENGER, R. M., SUGAWARA, Y., KUMAR, S., SALAPURA, V., SATTERFIELD, D. L., STEINMACHER-BUROW, B., AND PARKER, J. J. The IBM Blue Gene/Q interconnection network and message unit. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2011), SC '11, ACM, pp. 26:1–26:10.

[52] CHUNG, I.-H., WALKUP, R. E., WEN, H.-F., AND YU, H. MPI performance analysis tools on Blue Gene/L. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing* (New York, NY, USA, 2006), SC '06, ACM.

[53] CROVELLA, M. E., AND LEBLANC, T. J. Parallel performance prediction using lost cycles analysis. In *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing* (Los Alamitos, CA, USA, 1994), SC'94, IEEE Computer Society Press, pp. 600–609.

[54] CZECHOWSKI, K., BATTAGLINO, C., MCCLANAHAN, C., CHANDRAMOWLISHWARAN, A., AND VUDUC, R. Balance principles for algorithm-architecture co-design. In *Proceedings of HotPar'11* (Berkeley, CA, USA), USENIX Association, pp. 9–9.

[55] DALLY, B. Power, programmability, and granularity: the challenges of exascale computing. In *IEEE Parallel & Distributed Processing Symposium* (2011), pp. 878–878.

[56] DE VOS, M., GUNST, A., AND NIJBOER, R. The LOFAR telescope: System architecture and signal processing. *Proceedings of the IEEE 97*, 8 (2009), 1431–1437.

[57] DEN STEEN, S. V., PESTEL, S. D., MECHRI, M., EYERMAN, S., CARLSON, T. E., BLACK-SCHAFFER, D., HAGERSTEN, E., AND EECKHOUT, L. Micro-architecture independent analytical processor performance and power modeling. In *ISPASS* (2015), IEEE Computer Society, pp. 32–41.

[58] DING, C., AND ZHONG, Y. Predicting whole-program locality through reuse distance analysis. *SIGPLAN Not. 38*, 5 (May 2003), 245–257.

[59] EYERMAN, S., EECKHOUT, L., KARKHANIS, T., AND SMITH, J. E. A mechanistic performance model for superscalar out-of-order processors. *ACM Trans. Comput. Syst. 27*, 2 (May 2009), 3:1–3:37.

[60] EYERMAN, S., SMITH, J., AND EECKHOUT, L. Characterizing the branch misprediction penalty. In *2006 IEEE International Symposium on Performance Analysis of Systems and Software* (March 2006), pp. 48–58.

[61] FERDMAN, M., ADILEH, A., KOCBERBER, O., VOLOS, S., ALISAFAEE, M., JEVDJIC, D., KAYNAK, C., POPESCU, A. D., AILAMAKI, A., AND FALSAFI, B. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proceedings of the seventeenth*

international conference on Architectural Support for Programming Languages and Operating Systems (New York, NY, USA, 2012), ASPLOS XVII, ACM, pp. 37–48.

[62] FIORIN, L., VERMIJ, E., VAN LUNTEREN, J., JONGERIUS, R., AND HAGLEITNER, C. Exploring the Design Space of an Energy-Efficient Accelerator for the SKA1-Low Central Signal Processor. *International Journal of Parallel Programming 44*, 5 (2016), 1003–1027.

[63] FOG, A. The microarchitecture of Intel, AMD and VIA CPUs. An optimization guide for assembly programmers and compiler makers. http://www.agner.org/optimize/microarchitecture.pdf, 1996 (last update 2016).

[64] GAHVARI, H., BAKER, A. H., SCHULZ, M., YANG, U. M., JORDAN, K. E., AND GROPP, W. Modeling the Performance of an Algebraic Multigrid Cycle on HPC Platforms. In *Proceedings of the International Conference on Supercomputing* (New York, NY, USA, 2011), ICS '11, ACM, pp. 172–181.

[65] HARDAVELLAS, N., FERDMAN, M., FALSAFI, B., AND AILAMAKI, A. Reactive NUCA: Near-optimal Block Placement and Replication in Distributed Caches. *SIGARCH Comput. Archit. News 37*, 3 (June 2009), 184–195.

[66] HARDAVELLAS, N., FERDMAN, M., FALSAFI, B., AND AILAMAKI, A. Toward dark silicon in servers. *IEEE Micro 31*, 4 (July 2011), 6–15.

[67] HARING, R., OHMACHT, M., FOX, T., GSCHWIND, M., SATTERFIELD, D., SUGAVANAM, K., COTEUS, P., HEIDELBERGER, P., BLUMRICH, M., WISNIEWSKI, R., GARA, A., CHIU, G.-T., BOYLE, P., CHIST, N., AND KIM, C. The IBM Blue Gene/Q compute chip. *Micro, IEEE 32*, 2 (March-April 2012), 48 –60.

[68] HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture, Fourth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[69] HILL, M. D., AND MARTY, M. R. Amdahl's law in the multicore era. *Computer 41*, 7 (July 2008), 33–38.

[70] HOEFLER, T. Bridging performance analysis tools and analytic performance modeling for HPC. In *Proceedings of the 2010 conference on Parallel processing* (Berlin, Heidelberg, 2011), Euro-Par 2010, Springer-Verlag, pp. 483–491.

[71] HOEFLER, T., GROPP, W., KRAMER, W., AND SNIR, M. Performance modeling for systematic performance tuning. In *State of the Practice Reports* (New York, NY, USA, 2011), SC '11, ACM, pp. 6:1–6:12.

[72] HOSTE, K., AND EECKHOUT, L. Microarchitecture-independent workload characterization. *IEEE Micro 27*, 3 (2007), 63–72.

[73] INTEL CORPORATION. Intel Xeon processor E5-2697 v2 (30M cache, 2.70 GHz). http://ark.intel.com/products/75283/Intel-Xeon-Processor-E5-2697-v2-30M-Cache-2_70-GHz.

[74] INTEL CORPORATION. Intel Xeon processor E5-2697 v3 (35M cache, 2.60 GHz) specifications. http://ark.intel.com/products/81059/Intel-Xeon-Processor-E5-2697-v3-35M-Cache-2_60-GHz.

[75] JONGERIUS, R. LOFAR retrospective analysis - analyzing LOFAR station processing. Tech. rep., IBM Research, 2012.

[76] JONGERIUS, R., ANGHEL, A., MARIANI, G., DITTMANN, G., VERMIJ, E., AND CORPORAAL, H. Analytic multi-core processor model for fast design-space exploration (under review). In *IEEE Transactions on Computers* (2017), TC.

[77] JONGERIUS, R., MARIANI, G., ANGHEL, A., DITTMANN, G., VERMIJ, E., AND CORPORAAL, H. Analytic processor model for fast design-space exploration. In *Proceedings of the 33rd IEEE International Conference on Computer Design (ICCD)* (2015), ICCD'15.

[78] JONGERIUS, R., WIJNHOLDS, S. J., NIJBOER, R., AND CORPORAAL, H. An End-to-End Computing Model for the Square Kilometre Array. *IEEE Computer 47*, 9 (2014), 48–54.

[79] JOSE, J., POTLURI, S., TOMKO, K., AND PANDA, D. K. Designing scalable Graph500 benchmark with hybrid MPI+OpenSHMEM programming models. In *ISC'13*, vol. 7905 of *Lecture Notes in Computer Science*, Springer, pp. 109–124.

[80] JOSHI, A., PHANSALKAR, A., EECKHOUT, L., AND JOHN, L. K. Measuring benchmark similarity using inherent program characteristics. *IEEE Trans. Comput. 55*, 6 (June 2006), 769–782.

[81] KERBYSON, D. J., ALME, H. J., HOISIE, A., PETRINI, F., WASSERMAN, H. J., AND GITTINGS, M. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing* (New York, NY, USA, 2001), SC '01, ACM, pp. 37–37.

[82] KINSY, M. A., CHO, M. H., WEN, T., SUH, E., VAN DIJK, M., AND DEVADAS, S. Application-aware deadlock-free oblivious routing. *SIGARCH Comput. Archit. News 37*, 3 (June 2009), 208–219.

[83] KNÜPFER, A., BRUNST, H., DOLESCHAL, J., JURENZ, M., LIEBER, M., MICKLER, H., MÜLLER, M., AND NAGEL, W. The Vampir Performance Analysis Tool-Set. In *Tools for High Performance Computing*, M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, Eds. Springer Berlin Heidelberg, 2008, pp. 139–155.

[84] KODI, A. K., NEEL, B., AND BRANTLEY, W. C. Photonic interconnects for exascale and datacenter architectures. *IEEE Micro 34*, 5 (2014), 18–30.

[85] KUON, I., AND ROSE, J. Measuring the Gap Between FPGAs and ASICs. In *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays* (New York, NY, USA, 2006), ACM, pp. 21–30.

[86] LABARTA, J., GIRONA, S., PILLET, V., CORTES, T., AND GREGORIS, L. DiP: A parallel program development environment. In *Proc. of the Second International Euro-Par Conference on Parallel Processing* (London, UK, 1996), vol. II, Springer-Verlag, pp. 665–674.

[87] LAM, M. S., AND WILSON, R. P. Limits of control flow on parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture* (1992), ISCA '92, pp. 46–57.

[88] LATTNER, C., AND ADVE, V. LLVM: A Compilation framework for lifelong program analysis & transformation. In *Proceedings of CGO'04*, pp. 75–86.

[89] LEE, B. C., AND BROOKS, D. M. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2006), ACM, pp. 185–194.

[90] LEFURGY, C., WANG, X., AND WARE, M. Server-level power control. In *Proceedings of the IEEE International Conference on Autonomic Computing (ICAC'07)* (2007).

[91] LI, S., AHN, J. H., STRONG, R. D., BROCKMAN, J. B., TULLSEN, D. M., AND JOUPPI, N. P. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual International Symposium on Microarchitecture* (New York, NY, USA, 2009), MICRO 42, ACM, pp. 469–480.

[92] LOW, Y., BICKSON, D., GONZALEZ, J., GUESTRIN, C., KYROLA, A., AND HELLERSTEIN, J. M. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proc. VLDB Endow. 5*, 8 (Apr. 2012), 716–727.

[93] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of PLDI'05* (New York, NY, USA, 2005), ACM, pp. 190–200.

[94] MARIANI, G., ANGHEL, A., JONGERIUS, R., AND DITTMANN, G. Scaling properties of parallel applications to exascale. *International Journal of Parallel Programming 44*, 5 (2016), 975–1002.

[95] MARIANI, G., ANGHEL, A., JONGERIUS, R., AND DITTMANN, G. Classification of thread profiles for scaling application behavior. *Parallel Computing (PARCO) 66* (2017), 1 – 21.

[96] MARIANI, G., PALERMO, G., ZACCARIA, V., AND SILVANO, C. Desperate++: An enhanced design space exploration framework using predictive simulation scheduling. *IEEE Trans. on CAD of Integrated Circuits and Systems 34*, 2 (2015), 293–306.

[97] MCFARLING, S. Combining Branch Predictors (WRL Technical Note TN-36). Tech. rep., 1993.

[98] MINKENBERG, C., DENZEL, W., RODRIGUEZ, G., AND BIRKE, R. *End-to-End Modeling and Simulation of High- Performance Computing Systems.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 201–240.

[99] MINKENBERG, C., AND RODRIGUEZ, G. Trace-driven co-simulation of high-performance computing systems using OMNeT++. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques* (ICST, Brussels, Belgium, Belgium, 2009), Simutools '09, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), pp. 65:1–65:8.

[100] MOORE, S., CRONK, D., WOLF, F., PURKAYASTHA, A., TELLER, P., ARAIZA, R., AGUILERA, M. G., AND NAVA, J. Performance Profiling and Analysis of DoD Applications using PAPI and TAU. *Proceedings of DoD HPCMP UGC 2005, IEEE, Nashville, TN.* (2005).

[101] MURPHY, R. C., AND KOGGE, P. M. On the memory access patterns of supercomputer applications: Benchmark selection and its implications. *IEEE Trans. Computers 56*, 7 (2007), 937–945.

[102] MURPHY, R. C., WHEELER, K., BARRETT, B., AND ANG, J. Introducing the Graph 500. *Cray User's Group (CUG)* (2010).

[103] ÖHRING, S. R., IBEL, M., DAS, S. K., AND KUMAR, M. J. On generalized fat trees. In *Proceedings of the 9th International Symposium on Parallel Processing* (Washington, DC, USA, 1995), IPPS '95, IEEE Computer Society, pp. 37–.

[104] PADUA, D. *Encyclopedia of Parallel Computing.* Springer Publishing Company, Incorporated, 2011.

[105] PATEL, A., AFRAM, F., CHEN, S., AND GHOSE, K. MARSS: A Full System Simulator for Multicore x86 CPUs. In *Proceedings of the 48th Design Automation Conference* (New York, NY, USA, 2011), DAC '11, ACM, pp. 1050–1055.

[106] P.E. DEWDNEY, W. TURBER, R.BRAUN, J. SANTANDER-VELA, M. WATERSON, AND G.-H. TAN. SKA1 system baseline v2 description SKA-TEL-SKO-0000308. Tech. rep., SKA Organisation, November 2015.

[107] PESTEL, S. D., EYERMAN, S., AND EECKHOUT, L. Micro-architecture independent branch behavior characterization. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2015, Philadelphia, PA, USA, March 29-31* (2015), pp. 135–144.

[108] PODDAR, S., JONGERIUS, R., LEANDRO, F., MARIANI, G., DITTMANN, G., ANGHEL, A., AND CORPORAAL, H. MeSAP: A fast analytic power model for DRAM memories. In *Proceedings of Design, Automation and Test in Europe* (2017), DATE'17.

[109] PRISACARI, B., RODRÍGUEZ, G., HEIDELBERGER, P., CHEN, D., MINKENBERG, C., AND HOEFLER, T. Efficient task placement and routing of nearest neighbor exchanges in

dragonfly networks. In *The 23rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'14, Vancouver, BC, Canada - June 23 - 27, 2014* (2014), pp. 129–140.

[110] PRISACARI, B., RODRIGUEZ, G., MINKENBERG, C., AND HOEFLER, T. Fast pattern-specific routing for fat tree networks. *ACM Trans. Archit. Code Optim. 10*, 4 (Dec. 2013), 36:1–36:25.

[111] RÄCKE, H. Minimizing congestion in general networks. In *Proceedings of the 43rd Symposium on Foundations of Computer Science* (Washington, DC, USA, 2002), FOCS'02, IEEE Computer Society, pp. 43–52.

[112] REED, D. A., AND DONGARRA, J. Exascale computing and big data. *Commun. ACM 58*, 7 (June 2015), 56–68.

[113] ROHLFS, K., AND THOMAS, W. *Tools of Radio Astronomy.* Springer Berlin Heidelberg, 2004.

[114] ROMILA, A. *FPGA-based pre-processor for the Square Kilometre Array telescope.* M.Sc. thesis: Eidgenoessische Technische Hochschule Zurich, 2015.

[115] SATISH, N., KIM, C., CHHUGANI, J., AND DUBEY, P. Large-scale energy-efficient graph traversal: A path to efficient data-intensive supercomputing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC'12* (2012), vol. 7905, pp. 14:1–14:11.

[116] SHAO, Y. S., AND BROOKS, D. ISA-independent workload characterization and its implications for specialized architectures. In *Proceedings of ISPASS'13*, pp. 245–255.

[117] SHARAPOV, I., KROEGER, R., DELAMARTER, G., CHEVERESAN, R., AND RAMSAY, M. A case study in top-down performance estimation for a large-scale parallel application. In *Proceedings of PPoPP'06*, ACM, pp. 81–89.

[118] SHENDE, S. S., AND MALONY, A. D. The Tau Parallel Performance System. *Int. J. High Perform. Comput. Appl. 20*, 2 (May 2006), 287–311.

[119] SINGLA, A., GODFREY, P. B., AND KOLLA, A. High throughput data center topology design. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014* [119], pp. 29–41.

[120] SNAVELY, A., CARRINGTON, L., WOLTER, N., LABARTA, J., BADIA, R., AND PURKAYASTHA, A. A framework for performance modeling and prediction. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing* (Los Alamitos, CA, USA, 2002), SC '02, IEEE Computer Society Press, pp. 1–17.

[121] SRINIVASAN, A. Improved approximations for edge-disjoint paths, unsplittable flow, and related routing problems. In *, 38th Annual Symposium on Foundations of Computer Science, 1997. Proceedings* (1997/10/20/22 1997), IEEE, IEEE, pp. 416 – 425.

[122] STANDARD PERFORMANCE EVALUATION CORPORATION. SPEC CPU2006 benchmark description. http://www.spec.org/cpu2006.

[123] STANLEY-MARBELL, P. ExaBounds—Better-than-back-of-the-envelope Analysis for Large-Scale Computing Systems . Tech. rep., IBM Research – Zurich, 2012.

[124] SUZUMURA, T., UENO, K., SATO, H., FUJISAWA, K., AND MATSUOKA, S. Performance characteristics of Graph500 on large-scale distributed environment. In *Proceedings of IISWC'11*, pp. 149–158.

[125] TEICH, J. Hardware/software codesign: The past, the present, and predicting the future. *Proceedings of the IEEE 100*, Centennial-Issue (2012), 1411–1430.

[126] THOZIYOOR, S., MURALIMANOHAR, N., AHN, J. H., AND JOUPPI, N. P. CACTI 5.1. Technical Report HPL-2008-20, Hewlett Packard Laboratories, November 2008.

[127] UENO, K., AND SUZUMURA, T. Highly Scalable Graph Search for the Graph500 Benchmark. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing* (New York, NY, USA, 2012), HPDC '12, ACM, pp. 149–160.

[128] VAN NIEUWPOORT, R. V., AND ROMEIN, J. W. Correlating radio astronomy signals with many-core hardware. *International Journal of Parallel Programming 39*, 1 (2011), 88–114.

[129] WANG, H.-S., ZHU, X., PEH, L.-S., AND MALIK, S. Orion: a power-performance simulator for interconnection networks. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture* (Los Alamitos, CA, USA, 2002), IEEE Computer Society Press, pp. 294–305.

[130] WEINBERG, J., MCCRACKEN, M. O., STROHMAIER, E., AND SNAVELY, A. Quantifying Locality In The Memory Access Patterns of HPC Applications. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing* (Washington, DC, USA, 2005), SC '05, IEEE Computer Society, pp. 50–.

[131] YEH, T.-Y., AND PATT, Y. N. A comparison of dynamic branch predictors that use two levels of branch history. *SIGARCH Comput. Archit. News 21*, 2 (May 1993), 257–266.

[132] YOKOTA, T., OOTSU, K., AND BABA, T. Potentials of branch predictors: From entropy viewpoints. In *Proceedings of the 21st International Conference on Architecture of Computing Systems* (Berlin, Heidelberg, 2008), ARCS'08, Springer-Verlag, pp. 273–285.

[133] ZHAI, J., CHEN, W., AND ZHENG, W. Phantom: predicting performance of parallel applications on large-scale parallel machines using a single node. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2010), PPoPP '10, ACM, pp. 305–314.

[134] ZHONG, Y., SHEN, X., AND DING, C. Program locality analysis using reuse distance. *ACM Trans. Program. Lang. Syst. 31*, 6 (Aug. 2009), 20:1–20:39.

[135] Zyuban, V., Taylor, S., Christensen, B., Hall, A., Gonzalez, C., Friedrich, J., Clougherty, F., Tetzloff, J., and Rao, R. IBM POWER7+ design for higher frequency at fixed power. *IBM Journal of Research and Development 57*, 6 (2013), 1:1–1:18.