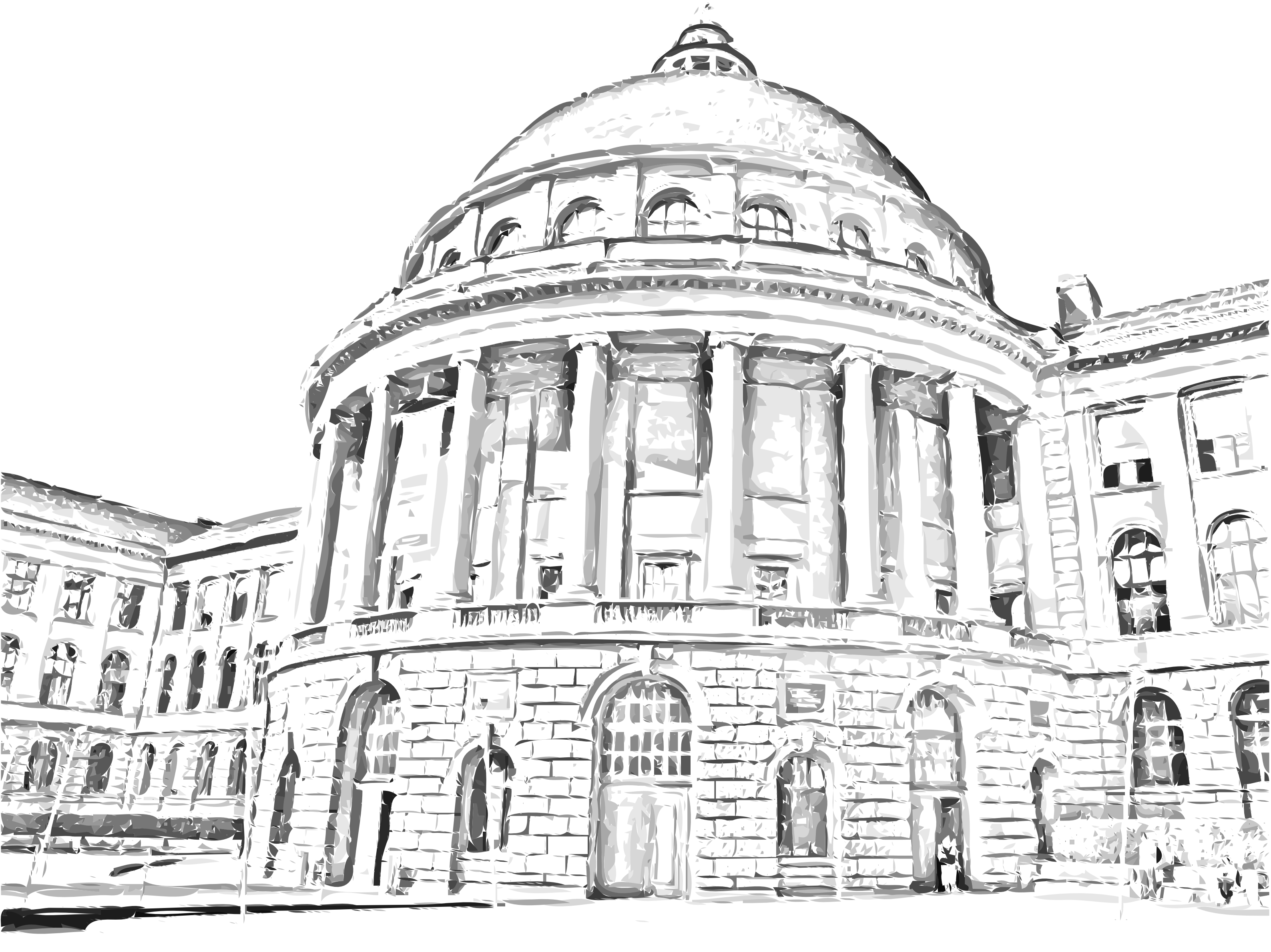
Temperature Aware Multiprocessing with Reliability Considerations

Devendra Rai



Diss. ETH № 22718

Diss. ETH No. 22718

Temperature Aware Multiprocessing with Reliability Considerations

A thesis submitted to attain the degree of

Doctor of Sciences of ETH Zürich
(Dr. sc. ETH Zürich)

presented by

Devendra Rai
M.Sc. University of Virginia, USA

born on 06.04.1981
citizen of India

accepted on the recommendation of

Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Jörg Henkel, co-examiner

2015

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

TIK-SCHRIFTENREIHE NR. 152

Devendra Rai

Temperature Aware Multiprocessing with Reliability Considerations

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

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

Diss. ETH No. 22718

Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Jörg Henkel, co-examiner

Examination date: May 29, 2015

To my wife, Shilpi.

You made this possible.

Our parents.

For your love, trust, and unconditional support.

My former German colleagues for the inspiration.

Elke Wilke, Ralf Rehbach, Andreas Goeckede, Georg Zöller, Jens Rudolph.

The greatest danger for most of us is not that our aim is too high and we miss it, but that it is too low and we reach it.

Attributed to Michelangelo in *The Element (2009)* by Ken Robinson

Abstract

With the proliferation of many and multicore processors in servers, desktops, laptops, and even mobile devices, we have become accustomed to expect ever greater computational performance from each new generation of such devices. For instance, as users of recent generation of mobile phones, we can now play games with rich graphics, enjoy rich multimedia, navigate the world, create and edit movies, none of which was possible only a few years ago. Similarly, our notebook computers are now as powerful as desktops of yesterday, allowing us to work effectively on the move.

The primary reason for such impressive growth in the power of computers has been continuous reduction in the size of the transistor, specifically its feature size. Transistors are fundamental building blocks of the state-of-the-art processors, and with smaller transistors, it is possible to pack more cores into the same die, resulting a multi- or a many-core processor. However, such a high level of integration is not without its own challenges.

The most significant challenge is the increased likelihood that such processors will overheat. This is primarily because the voltages required to operate such transistors have not reduced in the same proportion as the size of transistors used to build state-of-the-art processors. As a result, such processors can experience very high power densities, leading to temperature hotspots, with consequences both in the short and long terms.

In the short term, processors which run too hot automatically trigger built-in thermal protection mechanisms which slow the processor down. This causes unplanned, discernible, and in some cases, unacceptable degradation in the system performance. In the long term, rapid fluctuations in the temperature of the processor in time or in space may reduce its reliability.

In this thesis, we focus on overcoming reliability and performance challenges posed by state-of-the-art processors. In particular, design techniques are presented which can be used to avoid, tolerate, and recover from thermally induced faults. All concepts proposed in this thesis have been specifically designed for time and resource constrained systems, and therefore can be applied to designing reliable embedded and signal processing systems. In brief, major topics covered in this thesis are:

- A technique to avoid thermally induced faults by estimating offline, the temperature of the processor when it executes a given set of applications. Furthermore, an analytical technique to estimate the worst case temperature of the processor is also presented, which may be used to quickly eliminate those use-cases (i.e., combinations of applications and corresponding schedules) that may overheat the processor.

- A technique to tolerate (or mask) permanent faults in a manner which enables the system to continue to satisfy functional and timing requirements even after it has experienced one or more faults, and
- A technique to recover from faults in a manner which allows an application to migrate from a faulty processor (core) to a fault-free location, alongwith its context. Subsequent to migration, the application can continue to compute from the instant in time when the fault was detected. The secondary objective is to use this technique for dynamic load balancing across the available processors, which also helps avoid faults.

In practice, designing reliable multiprocessing systems may require the application of all three approaches simultaneously. Whereas the proposed fault avoidance technique is specific to thermally induced faults, proposed approaches for fault tolerance and recovery are comparatively general, and can be applied to protect a system from a broader class of faults. A highlight of this thesis is that all concepts have been validated through prototyping and extensive tests on multiple state-of-the-art multi- and many-core processors, such as the Intel Xeon family, Intel i7 family, and the Intel Single Chip Cloud (SCC) many-core processor.

Zusammenfassung

Mit der wachsenden Verbreitung von Viel- und Mehrkernprozessoren – sei es in Servern, Desktops, Laptops, ja sogar Smartphones – ist für uns ein Zuwachs an Rechenleistung mit jeder neuen Generation dieser Geräte selbstverständlich geworden. Als Nutzer der jüngsten Generation von Mobiltelefonen, zum Beispiel, können wir jetzt Spiele mit opulenter Grafik spielen, Multimedia in hoher Qualität genießen, rund um den Globus navigieren oder Filme erstellen und bearbeiten, was alles vor wenigen Jahren noch unmöglich war. Ebenso sind unsere Notebooks jetzt so leistungsfähig wie Desktops von gestern und erlauben uns so ein effektives Arbeiten auch unterwegs.

Der Hauptgrund für diese eindrucksvolle Steigerung der Leistungsfähigkeit von Computern ist die kontinuierliche Verringerung der Größe der Transistoren, genauer gesagt, ihrer Gatterlängen. Transistoren sind die grundlegenden Bausteine moderner Prozessoren, und mit kleineren Transistoren kann man mehr Prozessorkerne in den gleichen Chip packen und erhält so Mehr- oder Vielkernprozessoren. Eine solch hohe Integrationsdichte bringt jedoch ganz neue Herausforderungen mit sich.

Die bedeutendste Herausforderung ist hierbei die erhöhte Wahrscheinlichkeit einer Überhitzung der Prozessoren. Diese rührt daher, dass die benötigte Versorgungsspannung sich nicht im selben Maße reduziert wie die Größe der verwendeten Transistoren. Dadurch können in solchen Prozessoren sehr hohe Leistungsdichten auftreten, was zu lokal konzentrierten Hitzepunkten («Hotspots») führt, die wiederum kurz- und langfristige Konsequenzen haben.

Kurzfristig lösen zu hohe Prozessortemperaturen durch in den Chip eingebaute Schutzmechanismen eine Verlangsamung des Prozessors aus. Dies führt ungeplant zu einer wahrnehmbaren, in einigen Fällen inakzeptablen Verschlechterung des Systemverhaltens. Langfristig können schnelle Fluktuationen – sowohl zeitlich als auch räumlich – der Temperatur eines Prozessors dessen Zuverlässigkeit beeinträchtigen.

Die vorliegende Dissertation widmet sich der Bewältigung der genannten Herausforderungen bei Zuverlässigkeit und Leistung auf aktuellen Prozessoren. Insbesondere werden Konstruktionsverfahren vorgestellt, die angewendet werden können, um temperaturbedingte Störungen zu

---

Thanks a lot to Bernhard Buchli and Andreas Tretter for helping with the translation.

vermeiden, zu tolerieren und zu beseitigen. Alle in dieser Arbeit vorgeschlagenen Konzepte wurden speziell für zeitkritische und ressourcenbeschränkte Systeme entwickelt und können daher zur Entwicklung verlässlicher eingebetteter Systeme und verlässlicher Signalverarbeitungssysteme verwendet werden. Zusammengefasst sind die Hauptthemen dieser Arbeit:

- Ein Verfahren für die Vermeidung thermisch induzierter Störungen durch eine vorgängige Abschätzung der erwarteten Prozessortemperatur bei der Ausführung einer bestimmten Gruppe von Anwendungen. Zusätzlich wird eine analytische Methode zur Abschätzung der Worst-Case-Temperatur des Prozessors vorgestellt, welche angewendet werden kann, um die Anwendungsfälle (d.h. Kombinationen von Anwendungen und deren Ausführungszeitplänen), die den Prozessor überhitzen können, schnell zu verwerfen.
- Eine Methode, um dauerhafte Störungen so zu tolerieren (oder zu verbergen), dass das System weiterhin alle funktionalen und zeitlichen Anforderungen erfüllt, selbst nachdem ein oder mehrere Fehler aufgetreten sind.
- Eine Methode zur Störungsbeseitigung, indem eine Anwendung – zusammen mit ihrem Ausführungskontext – von einem fehlerhaften Prozessor(-kern) auf einen fehlerfreien umgesiedelt wird. Im Anschluss an die Migration kann die Anwendung an dem Punkt fortgesetzt werden, an dem die Störung erkannt wurde. Ein zweites Ziel ist es, diese Technik für eine dynamische Lastverteilung über alle verfügbaren Prozessoren anzuwenden, was wiederum Störungen vermeidet.

In der Praxis wird es manchmal notwendig sein, alle drei Ansätze gleichzeitig zur Anwendung zu bringen, um zuverlässige Multiprozessorsysteme zu erhalten. Während das vorgestellte Verfahren zur Störungsvermeidung speziell thermische induzierte Störungen betrifft, sind die vorgeschlagenen Konzepte für Fehlertoleranz und Störungsbeseitigung vergleichsweise allgemein und können für ein breiteres Spektrum von Störungen eingesetzt werden. Eine besondere Stärke dieser Arbeit ist, dass alle Konzepte durch Prototyping und umfangreiche Tests auf mehreren aktuellen Mehr- und Vielkernprozessoren validiert wurden, so z.B. der Intel-Xeon-Familie, der Intel-i7-Familie und dem Intel-Single-Chip-Cloud-Vielkernprozessor (SCC).

Acknowledgement

This thesis would not have been possible without the support of Prof. Lothar Thiele, who guided my efforts with his brilliance, time, patience, and generosity. I would also like to thank Prof. Jörg Henkel, the co-examiner for this thesis, for his time and constructive feedback.

I owe a lot to my colleagues, past and present, for supporting me in my work, and for the patience they have shown to me over the years. It has been a privilege to work with such bright colleagues at such a prestigious university. The most important lesson I take from here is of humility and pride, with pride stemming from the knowledge that I was always surrounded by brilliant but unassuming people, and humility because I am aware that I may never equal my colleagues in terms of excellence.

Most importantly, I am deeply indebted to my wife, Shilpi for her unconditional love and support during my time as a doctoral student. She encouraged me when I found it difficult to carry on, gave me hope when I was depressed, and for showing patience when I spent entire days, nights, and weekends working on a paper, a piece of software, or an experiment. She made it all possible. I am thankful to our parents for their patience, trust, and unequivocal support without which I would not have finished this thesis.

I would like to thank my former German colleagues with whom I worked during my times as an automotive embedded software engineer. It was here that I experienced excellence at work, which inspired me to return to the university to improve my skills.

Settling down in the German speaking city of Zürich, navigating legal regulations, and finding an apartment would not have been possible for an outsider like me without the support of Monica Fricker, a now-retired administrative assistant, and Beat Futterknecht, the current head of administration of the department.

The experiments reported in this work would not have been possible without the cooperation of our IT support staff, Thomas Steingruber, and Benny Gächter.

Finally, I would like to thank Tanja Lantz and Friederike Brütsch for helping me stay organized with administrative matters.

Contents

1

Introduction

This thesis presents techniques to avoid, tolerate, and recover from faults in resource- and time- constrained systems. The focus is on thermal faults, which are increasingly likely in state-of-the-art processors. In order to avoid thermal faults, a calibration based approach for constructing the thermal model of the given processor is presented. The thermal model is then utilized to estimate *apriori* the temperature experienced by the processor when it executes a given set of tasks. With the thermal model available, we follow up with an analytical technique to estimate the peak temperature that may be experienced by the processor, given an abstract description of the workload to be executed. Together, the thermal model and the peak temperature estimation technique are used to predict whether or not the processor will *ever* be in the danger of overheating and suffering from a thermal fault.

Next, in order to prevent any interruption in tasks and services in case a fault does occur, a fault detection and tolerance technique inspired by *N*-version programming is proposed. As several relatively mature approaches are available for detecting and tolerating value faults, this work focusses on detecting and tolerating timing faults in resource- and time-constrained systems. The proposed approach is specifically designed for process networks: a popular model of computation for tasks which operate on streaming data.

Finally, the problem of recovering from faults by task migration is considered. Again, the focus is on process networks and the challenge lies in correctly computing the context of processes which must migrate. In the context of resource- and time- constrained systems executing process networks, a new approach is presented wherein a process can guide itself into a so-called *stable state* in which its own context can be correctly collected. Post migration, the process can restore its context and continue computation at a new fault-free location.

The next section (i.e., Section 1.1) presents a background discussion on the causes and consequences of faults specifically considered in this thesis. Subsequently, we identify main challenges which must be addressed in order to achieve a reliable system in Section 1.3. The overall problem statement and a brief list of contributions made through this thesis is presented in Section 1.4, which is followed by a detailed plan of the thesis in Section 1.5.

1.1 The Other Side of Technology Scaling

It is widely acknowledged that steady improvements in the capability of processors have been driven largely by shrinking device geometries, also referred to as technology scaling. However, the voltages required to operate these devices have not scaled at the same pace as device geometries, in what is known as non-ideal CMOS scaling. A consequence of non-ideal scaling has been that the state-of-the-art processors can attain very high power densities and generate large amounts of heat which is difficult to dissipate using traditional cooling approaches such as a fan. As a result, it is increasingly likely that these state-of-the-art processors may experience very high operating temperatures, often near the critical temperature. Furthermore, data provided by Intel and the International Technology Roadmap for Semiconductors (ITRS) point to a future wherein it may not even be possible to execute applications on all the cores simultaneously due to the danger of fatally overheating the processor, an observation commonly known as *Dark Silicon*, see [EBSA$^+$11, HFFA11, SGM$^+$14].

In the long term, processors which either run too hot or experience rapid changes in temperature have reduced reliability as compared to the processors whose operating temperature has been carefully controlled. In the short term, the performance of tasks and services executing on a hot processor may deteriorate to an unacceptable level, and in extreme cases, hosted tasks and services may become unavailable if the processor shuts itself down to avoid any thermally induced damage. This inability of the processor to provide the expected computing performance specifically due to temperature issues described above is referred to as a *thermal fault*, which if not mitigated may cause the failure of the entire system executing tasks and services.

Another consequence of technology scaling has been to make processors more vulnerable to faults which may be triggered simply by exposure to the physical environment. That is, elements usually present in the environment such as cosmic particles and electromagnetic radiations are more likely to cause faults leading to errors in state-of-the-art processors. However, in contrast to faults caused by an overheated processor, these faults cannot be anticipated for, or avoided without significant and costly changes to the given hardware (e.g., radiation hardened processors).

1.2 The State-of-the-Art

The computing community along with the hardware vendors have attempted to mitigate the processor overheating problem using a combination of hardware and software approaches.

Software approaches attempt to control temperature by carefully controlling the workload being executed by the processor, either by shaping the incoming workload (e.g., by buffering some tasks), or by means of scheduling, see [KT11, FCWT09]. A more recent software oriented approach relies on selectively using cores in the given multi- or many-core processor which have special spatial characteristics, e.g., ensuring that no heat dissipating core (i.e., a *hot* core) has a hot neighbor, in what is known as *Dark Silicon Patterning*, see [SGHM14]. Yet another method is to migrate tasks between a set of similar processing elements in order to avoid overheating any given element, see [EAH12].

Hardware oriented solutions include capping of clock frequencies and integration of sophisticated power and temperature management features into the processor. However, in a bid to ensure that the every new generation of the processor performs better than the previous one (e.g., higher number of instructions per second, IPS), hardware vendors continue to pack increasing number of cores, which in turn increases the likeliness that the processor may overheat. As a result, processor architectures featuring novel devices, such as *Steep Slope Devices*, e.g., the Interband Tunnel FET (TFET) are being proposed. These devices can operate at voltages considerably lower than the CMOS transistors currently found in the state-of-the-art processors. Steep Slope Devices feature relatively low leakage currents, lower power densities, and also do not dissipate as much heat. Since the computational capacity of processing elements (e.g., cores) built out of such devices are expected is expected to be relatively limited as compared to CMOS based devices, researchers recommend heterogeneous processor architectures featuring processing elements that are built using conventional CMOS process, as well as TFET technology, on the same die, see [KSS$^+$12]. An operating system may dynamically choose the type of processing element to use (e.g., CMOS or TFET) depending upon application requirements, see [HNPT13].

1.3 Challenges

In general, techniques are already available that can be used to avoid, tolerate, and finally recover from such faults. However, the available approaches are limited in either one or more of the following aspects:

- The available approaches are not specifically designed for resource- and time- constrained systems, the class of systems which are specifically considered in this thesis; or
- The approaches seek to utilize processor architectures which are not yet mainstream, e.g., processor constructed out of Steep Slope Devices; or
- Solutions which seek to mitigate thermally induced faults rely on thermal models which are either too abstract to be reasonably accurate (e.g., lumped models) or are limited to dated processor architectures (e.g., hotspot simulator with the DEC-Alpha processor).

A wide class of automotive and medical electronics systems are resource- and time- constrained in nature. Specific examples systems which operate of streams of data (e.g., sensor values), such as infotainment systems, (automotive) collision avoidance systems, automotive engine management systems. Such systems are usually characterized by limited compute, network, and memory resources, but are expected to be reliable and provide services under tight timing bounds. For such systems, approaches for mitigating thermally induced faults such as those which require buffering tasks, or modifying application schedules may not be feasible. Tasks operating over streaming data are most commonly modeled as dataflow process networks (e.g., Kahn Process Networks) as these are simple to implement, self-scheduled, and provide natural support for expressing parallelism. However, as process networks feature asynchronously and autonomously executing processes, these present a significant challenge in the design of fault tolerance and recovery solutions.

Furthermore, this thesis considers commercially available state-of-the-art processors in order to develop fault tolerance, avoidance, and recovery solutions, with an emphasis on thermally faults. Therefore, the proposed solutions must be easily implementable on commercial state-of-the-art processors, with a specific focus on resource constrained systems. To this end, we exploit the difference in mechanics of thermal faults as compared to those faults which are hard to anticipate, and may lead to sudden loss of functionality of the processor, e.g., faults induced by radiation. In general, it is possible to anticipate and avoid thermal faults *if* a thermal model of the given processor can be so constructed that it allows apriori estimation of temperature traces on a given *state-of-the-art* processor when it executes a given set of tasks, in a reasonable time and with reasonable accuracy. The strength of this approach is that it does not depend on a yet-experimental new device technology (e.g., TFET), or require specifically designed processors (e.g., processors with fluid-based cooling), and hence is widely applicable. Faults which are difficult to anticipate (e.g., those induced by radiation) are tolerated, and finally can be recovered from.

The following sections present more details on the challenges associated with avoiding, tolerating and recovering from faults, in the context of resource- and time- constrained systems. Specifically, three main challenges have been identified which motivate the work presented in this thesis:

1.3.1 Avoiding Thermal Faults

Limitations of Current Solutions

In general, techniques to prevent processor overheating are already available, and some of these solutions are already found in the commercial state-of-the-art processors. One popular technique is to dynamically lower the processor's clock speed and/or voltages (DVFS) as the processor gets too hot. This has the effect of slowing down the processor which also lowers its temperature. However, this approach is usually reactive in nature resulting in an unforeseen and unplanned loss of performance. If the system operates under tight timing constraints, any performance degradation may have serious consequences on the overall reliability of the system. For instance, an automotive engine management computer may be processing several live streams of sensor data for computing critical operating parameters (e.g., when to fire a given spark plug). Obviously, there are timing constraints under which the streams must be processed and therefore, any unforeseen degradation in the available computing capacity may not be acceptable due to performance and safety considerations. In addition, if the system uses a resource-constrained processor, the additional compute burden imposed by dynamic temperature management solutions may be unacceptable. In addition, approaches relying on buffering tasks for limiting processor temperature may not be feasible.

In contrast, *if* it is possible to accurately estimate *at design time* how the temperature of the processor will evolve when it will execute a given set of tasks, then it may be possible to develop thermally-optimized task binding (i.e., mapping of tasks to cores, their respective schedules, processor clock frequency, and required cooling) options which ensure that the processor *never* overheats at runtime, in turn avoiding any unforeseen performance losses. As indicated earlier, solutions which rely on scheduling decisions to avoid processor overheating are already available, but often rely on abstract, or dated thermal models. This design-time search for optimal task bindings is referred to as Design Space Exploration (DSE). Since the approach obviates the need for any runtime thermal management, it does not add any computational burden on an already time- and resource- constrained system.

Challenges in Offline Estimation of Temperature Traces

In general, estimating the time-trace of temperature of a given processor at design time is a complex problem as it requires accurately accounting for all the factors which influence the overall temperature of the processor, such as the technology node, available cooling, number of cores, tasks, processor housing and the ambient temperature. A wide spectrum of approaches are available, each differing from the other in the amount of information required for computing temperature traces, associated accuracy, and the speed of computation. On one end of the spectrum we have so called *lumped models* which approximate the entire processor as a point source of heat. The high level of abstraction makes such models computationally fast on one hand, and erroneous on the other. The other end of the spectrum consists of fine-grained numerical simulators, such as Hotspot. These simulators require lot of details to be known about the processor (e.g., exact floor-plan, power trace for each micro-architectural unit inside the processor, cooling model to name a few) and are considered to be accurate. However, numerical simulators are computationally intensive, and computing each new temperature data point requires solving large number of system equations. In case of a state-of-the-art processor with multiple cores, the time required to compute a temperature trace consisting of thousands of data points may be simply too long to be feasible in a practical Design Space Exploration use case wherein hundreds of temperature traces may need to be computed in search for an appropriate solution.

A New Approach: Calibration based Thermal Models

We propose a middle ground between highly coarse-grained lumped models, and highly fine-grained numerical simulators. We build a thermal model of the given processor by a sequence of calibration experiments conducted on the processor of interest. The proposed technique does not require the system designer to have accurate knowledge of possibly several hard to get parameters, which are otherwise critically required by numerical simulators such as Hotspot. Many of these parameters are rarely known with sufficient accuracy to anyone except the vendor itself, making it difficult to use numerical simulators for building thermal models of state-of-the-art processors. An additional advantage of the proposed approach is that the time required to compute several thousands of temperature data points is in the order of seconds, which may be considered acceptable for Design Space Exploration. The temperature traces computed using the new approach are also accurate. Experimental results show that the error in the estimated traces is in the same order as the noise in the temperature sensors.

1.3.2 Fault Tolerance in Resource Constrained Systems

Limitations of Current Solutions

As expected, several general approaches for tolerating faults are already available, which include transparently masking faults at the hardware level, or by hardening processors (e.g., radiation hardened processors), see [HBZ$^+$14] for a discussion on various techniques available to improve reliability of a computer-based system.

Software oriented solutions include generation of multiple executable copies (henceforth referred to succinctly as *replicas*) of the same application but with design diversity to be executed simultaneously on the processor, a technique known as *N-version programming*. Such executables avoid common-mode faults. Compilers can also be used to generate different executables that execute on different elements of the processor, thereby ensuring that failure of one (or more) processing elements does not affect all copies of the application.

In case a fault causes a replica to output incorrect results, it may be necessary to detect such a fault in order to isolate the faulty replica to prevent the system from producing incorrect results. Various mechanisms have been proposed to detect a potential value fault in one or more of the replicas, and also to arbitrate between their respective outputs for computing the final output. The simplest of the arbitration schemes is the majority voting, whereas more sophisticated approaches may combine probabilistic and heuristic analysis in order to compute the final output.

In systems where timing is critical, replicas may exhibit *both* timing as well as value faults. Timing faults are observed when a replica provides the correct output in terms of value but not in terms of timing, i.e., the output is not computed within the required time bounds. If the task exhibits simple timing properties, such as periodicity, then simple timer based fault detection approaches may be used, e.g., watchdogs. Detection of timing faults is much more challenging if tasks operate under more complex timing patterns. Various approaches to detect timing faults under such conditions have been proposed. One may use a set of timers which timeout at different intervals, and an indication of a timing fault may be construed from the state of the selected timers. However, solutions which depend on several alarms are not scalable in terms of resources used, especially when the timing complexity and the number of tasks to be monitored are significant. One may also detect timing faults based on the analysis the timestamps of data packets (or tokens) received within some limited time-window in the immediate past. Based on a set of high-level timing properties supplied by the designer, the approach checks the conformance of timestamps of the received packets to the given properties. Accordingly, the approach requires at *least* one system timer, dedicated memory for storing timestamps, as well as computational resources for runtime timing analysis. More complex approaches based on a system of state observers have also been proposed. However, such approaches may be infeasible for use in time- and resource- constrained systems due to their high resource demands, and also require specifically crafted tasks that support such a system of state-observers.

Challenges in Detecting and Tolerating Timing Faults in Constrained Systems

Timing jitters and occasionally bursty outputs are a common characteristic of process networks. In case of an automotive engine management system, jitters and bursts may be rare (and may be the consequence of a fault), whereas in multimedia process networks, these are expected, and normal. Detecting and tolerating timing faults is especially challenging in such cases as any feasible solution must be able to distinguish between between normal and tolerable timing variations and those which indicate a fault. Furthermore, it is usually the case that a fault tolerant system requires tasks be specifically written to support the chosen fault detection and tolerance strategy. For instance, it is common practice that tasks must reset a watchdog at regular intervals as a demonstration of their fault-free status. However, in cases of legacy and/or complex process networks, a redesign to suit the chosen fault detection and tolerance strategy may not be feasible. In summary, detection and tolerance of timing faults in time- and resource- constrained systems remains an open challenge.

A New Approach: Using FIFO buffers for Detecting and Tolerating Timing Faults

We propose to use network and real time calculus to translate the difficult problem of detecting timing faults into a much easier problem of counting the number of data tokens in a set of carefully designed FIFO buffers. In addition, we derive a fault tolerance solution at no extra cost to the system resources. Since the approach involves merely counting, the overall solution is lightweight both in memory and computational overhead imposed on the system. The proposed technique requires abstract information on the expected interface-level timing properties under which the system operates, which are usually available at design time, and therefore, does not impose any additional burden on the system designer. Lastly, the proposed approach treats the given process network as a black-box, does not require any modifications to their design, thereby making the proposed approach applicable for use with large, complex, and legacy tasks.

1.3.3 Fault Recovery and Load Balancing

Limitations of Current Solutions

Both fault recovery and load balancing may require that tasks be moved between the processor(s). In one use-case, tasks may be moved to different processor (cores) in order to load-balance the processor, or to ensure that the no part of the processor gets too hot. Another use-case is recovering from faults, wherein a task may need to be moved to a new fault-free core (or a processor) so that computation can continue. Most current approaches for moving tasks (or, task migration) are either resource heavy, restricted to shared memory systems, or require tasks and the operating system to explicitly support migration. For instance, task migration requires that the context of the affected task be collected, which is then restored to it post-migration. A possible solution is to design tasks which store their context into a remote memory location at regular intervals (i.e., checkpointing), although it may be possible that the operating system takes over the burden of checkpointing tasks. Thus, it is feasible to migrate tasks using a custom operating system, it is nonetheless restricted to processors which have enough resources to support such an operating system. It is also easy to see that checkpointing may consume significant compute, network, and memory resources if tasks have significant amount of context data to be stored (e.g., tasks are data intensive, and/or have deep states).

In contrast, time- and resource- constrained systems seldom have spare memory, compute capacity, or network bandwidth to support regular checkpointing. Also, such constrained systems feature lean operating systems (e.g., microkernels) which may not capable of supporting task migration.

Challenges in Task Migration Under Resource Constraints

As previously pointed out, a custom operating system based approach to task migration is restricted to systems which can support such an operating system. A major challenge in designing a solution for task migration specific to process networks stems from the nature of the process network itself: processes execute autonomously and asynchronously with no clear "master" or "controller" process. In such cases, the context of each process continues to change not only as the it executes, but also as it exchanges data between its parents and children. Therefore, in order to correctly collect the context of a process, all the parents must be notified of an impending migration of their child so that parents do not transmit new data to the affected process. The execution of process(es) to be migrated must be suspended, and all data transmitted by their respective parents must be accounted for in the context. In summary, the set of processes designated for migration must be guided into a *stable state* in which their respective contexts will not change any further. The asynchronous nature of the process network makes it difficult to estimate a bound on the delay that a process may experience before it has received all packets transmitted to it. Process networks implemented on a distributed memory system further complicate this problem, as in this case, a process may need to additionally account for network delays to receive any in-flight packets before it can store its context. As already discussed, processes can checkpoint themselves at regular intervals so that context information is readily available if migration is required, but such an approach may impose severe memory and computational overhead for complex process networks.

A New Approach: Stabilizing Process Networks Under Resource Constraints

Much of the compute and memory overhead associated with task migration is due to the implicit assumption that a faults can lead to abrupt loss of functionality, and hence the need for expensive and regular checkpointing. However, thermal faults need not be sudden: a system can be designed to allow an advanced warning as the temperature of the system crosses a pre-determined threshold but has not yet overheated, and hence the system functionality is available, at least for a limited time. It is only when the overheat warning is received, task migration procedure can be triggered, thus avoiding costly checkpointing when the system operates within normal conditions. In addition, the structure of dataflow process networks offers further cost optimization opportunities in terms of memory: a process is normally required to buffer input data before processing. When the process context needs to be computed, these input buffers and the data contained therein can be simply moved into the collected context. Furthermore, keeping with the asynchronous and autonomous character of process networks, we propose an approach in which each process can autonomously decide to migrate, and subsequently stabilizes itself in a local co-ordination with its parents and children. The emphasis is placed on an efficient stabilization procedure, whereas the required memory and compute efficiency flows from the architecture of the process network itself. The proposed approach is correct, i.e., the functionality of the process network before and after stabilization is preserved. Furthermore, the proposed stabilization procedure is independent of any delay, either in the operating system, the processor, the memory, or the network.

1.4 Problem Statement and Contribution

Brief Problem Statement

Following the discussion, the problem solved in this thesis can be briefly summarized as:

Given a processor susceptible to thermal faults with possibly many cores, design a system which avoids, tolerates and recover from faults such that the given resource- constrained system keeps operating within the specified timing constraints.

Brief Summary of Contributions

Specifically for process networks executing on time- and resource- constrained systems, this thesis makes the following contributions to the state-of-the-art:

- A new calibration based technique for constructing an accurate and computationally efficient thermal model of the given processor to be utilized for designing systems that avoid thermal faults;
- A new formal analytical technique to estimate peak temperature of the processor given any task and the associated workload description, again to be used for avoiding thermal faults;
- A new computationally efficient fault-tolerance technique;
- A new event triggered technique which prepares an *executing* process network for migration from a faulty processor to a non-faulty one, thereby enabling a given system to recover from faults; and
- All presented techniques have been validated by extensive experiments on a variety of state-of-the-art processors.

1.5 Thesis Overview and Contributions

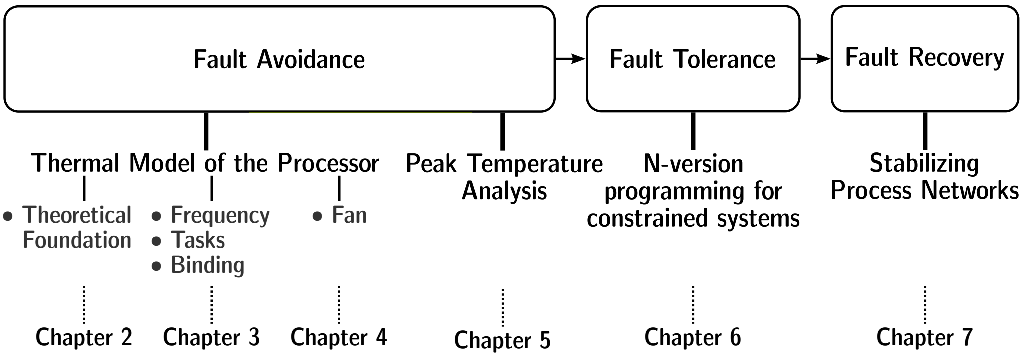


Figure 1.1: Organization of this thesis.

In the context of thermal faults, this thesis presents new techniques to avoid, tolerate and recover from such faults. All techniques presented in this thesis have been developed specifically for time- and resource- constrained systems. The organization of this thesis is summarized in Figure 1.1, whereas detailed contributions of each chapter are presented next:

Chapter 2

This chapter presents the models and methods used for constructing the thermal model of a given processor. The proposed method exploits the predictable runtime behavior of tasks designed for embedded systems (succinctly, *embedded tasks*) for constructing the thermal model of the system. The proposed approach does not require any hard-to-get information such as the processor floorplan, or detailed power traces. These models and methods are later exploited in chapters 3 and 4 for constructing thermal models for state-of-the-art systems. Specifically, this chapter builds on the following ideas:

1. We show that the runtime behavior of embedded tasks is (often) deterministic in the following sense: when the given task executes, then the sequence of instructions executed, and their relative proportion remain similar (i.e., the same order) irrespective of inputs, and across different invocations of the given task;
2. As a result, the total power consumed by a given embedded task is (approximately) constant over time, and is largely determined by the task itself. Furthermore, the distribution of the total power between different micro-architectural entities of the processor (core) is also (approximately) constant;
3. This consistency in the runtime behavior of tasks causes predictable changes in the temperature of the processor which can be captured with sufficient accuracy into a thermal model;
4. It is possible to correctly estimate the temperature trace of a processor from the estimated thermal model together with the task binding information (i.e., its schedule, mapping to core, processor clock speed, cooling applied); and
5. Construction of the thermal model does not require any hard-to-get information about the processor such as the detailed processor floorplan, or accurate power traces at the granularity of micro-architectural units, which have traditionally been considered a critical pre-requisite.

Chapter 3

This chapter uses principles presented in chapter 2 in order to construct a thermal model for a state-of-the-art Xeon 8-core processor. Also proposed is a method to account for the effect of temperature and power management algorithms found in the processor.

The chapter makes the following contributions to the state-of-the-art:

1. A calibration based technique for constructing a thermal model of a given state-of-the-art processor;
    - Only on-board temperature sensors are used,
    - The influence of on-chip temperature and power management algorithms on the temperature of the processor is also also considered.

2. Extensive experiment on the Intel Xeon 8-core processor show that the error in the temperature traces computed using the model is no greater that the quantization error in the on-chip temperature sensors.

This, the results presented in this chapter may be viewed as a (partial) validation of the techniques proposed in Chapter 2.

Chapter 4

This chapters presents a calibration based technique to incorporate the effect of the system fan into the thermal model of the processor. The result is a thermal model of the system fan which can be cascaded to already available fan-unaware system thermal models. To this end, the chapter presents the following main concepts:

1. A calibration based technique for constructing a cascadable thermal model of the system fan;
    - The model is constructed solely from the data obtained during calibration experiments,
    - The technique obviates the need for detailed power traces traditionally considered critical for constructing the thermal model of a fan.
2. The resulting thermal model of the fan depends only upon the speed of the fan and is independent of all other factors;
3. Extensive experiments on a state-of-the-art notebook computer demonstrate the accuracy of the thermal model.

Together with Chapter 3, this chapter fully validates the ideas proposed in Chapter 2, i.e., it is possible to construct an accurate and computationally efficient thermal model of the complete computer *system* from observing temperature traces when the processor executes a set of calibration experiments. Though the proposed approach is restricted to tasks which have deterministic runtime characteristics, it is nevertheless applicable to a broad class of embedded and signal processing systems. The same approach has been successfully applied to estimate the thermal model of the fan, but the result is more general: the thermal model of the fan holds irrespective of the class (or type) of tasks that may execute on the processor.

Chapter 5

This chapter presents a formal method based on network and real time calculus for estimating the worst case temperature of a processor, given the thermal model of the processor and information about the workload. It is known that even exhaustive simulations may not be able to accurately yield such worst case estimates making the formal approach cost efficient. The worst case temperature estimates may then be used to design better binding solutions, or to configure the system for tolerating and recovering from any inevitable thermal faults. This chapter covers the following new concepts:

1. A formal method based on network and real time calculus is used to estimate worst case temperature of a processor, given a thermal model and a description of the workload;
2. It is shown that the obtained estimates are tight; and
3. A method to compute the associated worst case workload (trace) is also presented.

Chapter 6

This chapter presents an *N*-version programming technique which is adapted for detecting and tolerating timing faults time- and resource- constrained systems. The proposed approach does not require the use of any timer resources, and is thus scalable. Fundamentally, it is shown that formalisms of network and real time calculus can be used to convert the difficult timing fault detection problem into a much simpler counting problem.

This chapter covers the following items:

1. The design of the replicator and the selector processes which are used for implementing the fault tolerant system;
2. Concepts from network and real time calculus that are exploited which allow detection of timing faults from observing fill level of various first in, first out (FIFO) buffers; and
3. Experimental results of a prototype implemented on the Intel Single Chip Cloud (SCC) processor confirm the scalability and resource efficiency of the proposed fault detection technique.

Chapter 7

This chapter focuses on a major problem encountered when attempting to migrate asynchronous process networks: upon receiving an event trigger, guide a set of asynchronously and autonomously executing processes into a stable state so that their contexts can be collected. In contrast to earlier results which have either discussed the problem of task migration from a theoretical standpoint or under simplifying assumptions, this chapter also discusses the implementation and results of the proposed technique on a state-of-the-art Intel SCC processor.

The focus of this chapter is on the following:

1. An event-triggered mechanism which guides a set of processes into a stable state wherein the associated contexts can be computed;
    - The presented technique is fully event driven and is therefore immune to various delays, jitters and other timing variations commonly observed when an asynchronous process network executes on a system-on-chip,
    - The computation and subsequent restore of the context to the processes is based on a prototype implemented using protothreads,
    - Case studies using the Intel SCC processor confirm both the correctness and the resource efficiency of this technique.

2

Theoretical Foundation: Construction of Thermal Models

Summary

This chapter presents methods and models to construct the thermal model of a given processor. The proposed approach does not require the system designer to know hard-to-get parameters about the processor of interest. Instead, a sequence of calibration experiments on the processor extracts all the parameters necessary for constructing an accurate and computationally efficient thermal model. The approach requires access to on-chip temperature sensors for calibration experiments. Furthermore, the proposed approach does not require any physical access to the processor of interest, which can be exploited for commercial applications.

2.1 Introduction

The most effective measure for avoiding thermal faults is to know apriori whether or not a given processor will overheat when it executes a known set of applications. Answering this question requires the capability to accurately estimate how the temperature of the processor will evolve at runtime when it executes the given set of applications. Thus, the fundamental requirement for avoiding thermal faults is an *accurate* and *computationally efficient* thermal model of the given processor. In addition to answering the aforementioned question, the thermal model can also be used to explore design solutions which meet a set of thermal constraints, e.g., how should the applications be mapped to the processor, how should their respective schedules be designed. Such thermal constraints may range from simple to complex. A simple thermal constraint may require the temperature of the processor to be always below a pre-defined threshold, whereas more complex constraints may limit the frequency and amplitude of thermal variations experienced by the processor.

However, constructing an efficient and accurate thermal model of the processor remains a challenge as many factors which influence the overall temperature of the processor have to be accurately accounted for. Some of these factors are cooling, processor clock speed, the given application set, mapping of applications to the processor, their respective scheduling information, technology node, processor floorplan, and system chassis. Though some of these parameters are easy to acquire, certain other information such as the detailed processor floorplan is rarely available. This chapter shows that *if* it is possible to restrict the *class* of applications to the embedded type, then it is possible to construct an accurate and computationally efficient thermal model relatively easily. The approach relies on access to on-board temperature sensors, and does not require the knowledge of any other hard-to-get information. A specially designed set of calibration experiments automatically extracts all the parameters necessary for the thermal model. An added benefit of the proposed approach is the possibility of constructing a thermal model of the given processor *without* requiring physical access to it, which may be exploited for commercial applications.

2.2 The Problem and Related Work

Several approaches are already available for constructing the thermal model of the processor, each differing from the other in accuracy as well as the speed at which the temperature traces (i.e., time ordered sequence of temperature measurements or estimates) can be computed. These approaches fall between two extremes: very accurate but computationally intensive numerical simulators on one hand, and abstract but computationally efficient models on the other.

2.2.1 Temperature Estimation Using a Simulator

One example of a state-of-the-art thermal simulator is Hotspot, see [HSS$^+$04]. Hotspot considers processors with traditional cooling systems (i.e., with or without a fan), whereas more recent thermal simulators are able to account for fluid cooling inside a processor, see [SVAB13]. Such simulators can be very accurate if all the necessary parameters can be supplied with high accuracy. Whereas some of the parameters (e.g., the application set, mapping of applications to cores, associated schedules, processor clock speed) may be easy to find, certain other details such as the detailed processor floorplan, power model or traces at the micro-architectural granularity, details of the cooling solution used, or the thermal characteristics of the processor may not be readily available and may even be very difficult to acquire. In case several processor architectures have to be investigated, the challenge in acquiring all relevant parameters for each architecture may quickly become overwhelming. Another (lesser) challenge is that numerical simulators estimate temperature traces by solving a large number of system equations, which may lead to significant computing times especially when dealing with state-of-the-art multicore processors. Thus, the difficulty in acquiring all the necessary parameters together with the compute intensive nature of numerical simulators make them infeasible in certain use cases (e.g., Design Space Exploration, DSE) wherein a hundreds, if not thousands of different thermal simulations may have to be performed, for each architecture under investigation.

Various approaches to overcome some of these challenges have been proposed. One approach is to restrict the processor under investigation to (an old) DEC Alpha 21264 model. The exact floorplan of this processor is widely available, and is also distributed with the Hotspot simulator package. The required power trace information is extracted from the Wattch simulator, thus completing all the necessary requirements for estimating temperature traces using Hotspot, see [BTM00]. A synthetic multicore processor constructed by scaling and tiling the original Alpha 21264 processor flooplan is already available for use with Hotspot, see [RV09]. An obvious restriction of this approach is that it is fundamentally limited to a dated processor model.

There have been attempts to acquire some of the hard-to-get parameters using indirect means. One approach proposes capturing heat maps from the silicon layer of the processor as it executes instructions, and subsequently use heuristic algorithms to back-estimate the detailed power model from captured heat maps. The estimated power model is then used with Hotspot for thermal simulations, see [MMNBR07]. However, the details of the processor floorplan must still be guessed, which in itself is a significant challenge as floorplans of the state-of-the-art processors are quite complex. Furthermore, the approach requires peeling off the packaging of the processor which may significantly alter its thermal properties. Therefore, it is not clear whether the technique and any of the associated results may be applicable to processors of the same make, but which retain their original packaging.

Another approach is to use event-counters as a proxy for power dissipation at the core level, see [CS06]. The associated power traces for use with Hotspot are computed using a simple polynomial in the selected event counters. The polynomial function itself is estimated by regression analysis of the observed temperature traces and the selected event-counters. The reported approach is simple, fast, and is able to compute correct steady-state temperatures, but shows significant inaccuracy in estimating the transients, specifically at the task-scheduling boundaries. In addition, the proposed approach requires processors which feature event or performance counters.

2.2.2 Temperature Estimation Using Lumped Models

One may get around the problem of having to acquire the knowledge of so many details about the processor by using so-called *lumped thermal models*. These models usually assume that the entire processor is a point source of heat, and in case of a multicore processor, cores are connected to each other via a simple resistor-capacitor network, see [WR10, GD05, RU08]. Lumped models are computationally efficient (i.e., can compute temperature traces very quickly) but may be very erroneous as lot of factors which impact the temperature of the processor are abstracted out, e.g., the effect of non uniform power distribution within the processor.

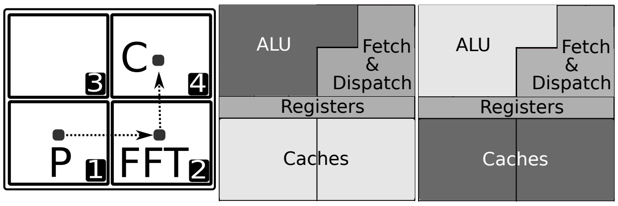


Figure 2.1: Producer (P), FFT and Consumer (C) applications run on a CMP. The power density distribution in micro-architectural components of the cores 1 and 2 is shown, where lighter shades show areas with higher power consumption.

As a specific example, consider a synthetic four-core chip-multiprocessor (CMP) executing three applications: *producer*, *FFT* and *consumer*, see Figure 2.1. The *producer* is in charge of creating data for the *FFT* application, which in turn supplies the results to the *consumer* application for display. The *producer* and *consumer* are I/O intensive applications, and it can be expected that a considerable amount of power is consumed in data caches. On the other hand, *FFT* is a compute-intensive application and the ALU will dominate the power consumption of the corresponding processor core. All three applications consume the same total power. The floorplan on the left shows the power density due to the FFT application executing on core 2, whereas the floorplan on the right shows the power density due to the producer and the consumer applications executing on cores 1 and 4. The temperature sensor is located near the upper left corner of the processor. hence, it is more sensitive to the heat generated by the computational units of the processor.

According to frequently used coarse grained temperature estimation techniques using lumped thermal models, the temperature of a core i is estimated as a function $\mathcal{F}$ of the total power consumed by the cores on the processor:

$$T_i(t) = \mathcal{F}_i\left(\mathsf{P}_1(t), \ldots, \mathsf{P}_n(t)\right) \tag{2.1}$$

where $\mathsf{P}_i(t)$ denotes the trace of total power consumption of a core i in the CMP, recorded at some given time resolution t_s from time 0 to t. In other words, lumped models ignore the spatial distribution of power within a core of the processor. We emulate the lumped model using the Hotspot simulator as follows: First, extract the detailed power trace information for each application by using the Wattch/Simplescalar tool chain, see [BTM00]. We assume a 4-core synthetic multicore processor based on the Alpha 21264, which is also supported by Hotspot, see [RV09]. Next, from the available detailed power trace information, we compute the trace of total power consumed by a given application, and distribute this uniformly between all the micro-architectural elements in the core which executes the given application. The resulting temperature traces for cores 1, 2, and 4 are shown in Figure 2.2. Temperature influence from cores to their neighbors has been accounted for. All applications run according the same schedule: tasks execute in lock-step, i.e. cores 1, 2, and 4 have the same total power trace over time. Communication between cores is implemented using FIFO buffers. Notice that the temperature traces for cores look similar, with core 2 showing a slightly higher temperature which can be attributed to having two hot neighboring cores.

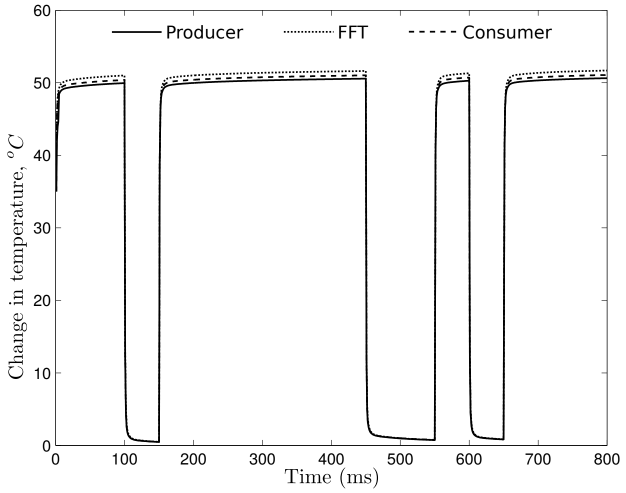


Figure 2.2: Estimated temperature traces on cores 1, 2, and 4 due to *Producer, Consumer* and *FFT* applications. Power is uniformly distributed between all micro-architectural elements in the respective cores to simulate a lumped model.

The experiment was repeated, but this time, the distribution of total power to the micro-architectural elements in the processor was made according to the results reported by the SimpleScalar and Wattch simulators.

In other words, the power distribution supplied to the Hotspot simulator for computing temperature traces is realistic. The resulting temperature traces are shown in Figure 2.3. Notice the difference in the temperature for cores which run the applications. Specifically, compared to traces in Figure 2.2, core 2 shows a significantly higher temperature as the FFT stresses the ALU of the processor which is nearer to the temperature sensor. The temperature for cores 1 and 3 is comparatively lower as most of the power is consumed in large caches which are further away from the temperature sensor as compared to the ALU region.

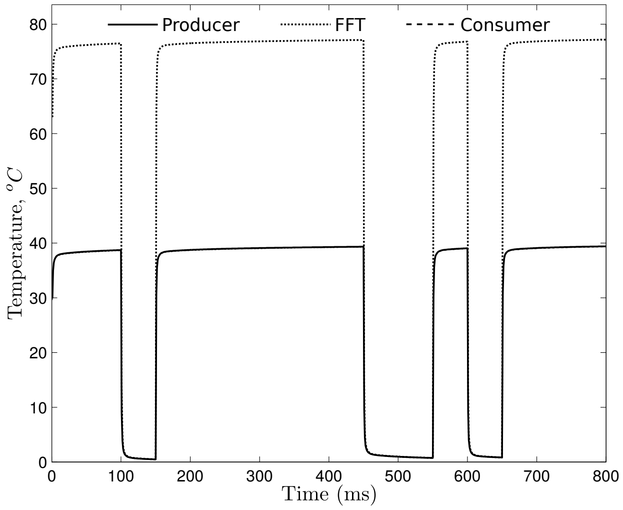


Figure 2.3: Estimated temperature traces on cores 1, 2, and 4 due to *Producer, Consumer* and *FFT* applications. Corresponding power distribution is as reported by SimpleScalar and Wattch.

The presented example assumes that the location of the temperature sensor is carefully selected by the vendor of the processor, and the temperature readouts provided by the sensors can be taken as a representative temperature of the core. Implicitly, the location of the thermal sensors suggests the regions of the processor which may be more vulnerable to temperature hotspots, and therefore, must be monitored more closely. However, lumped models abstract out spatial distribution of the total power in the core of the processor, which may lead to large errors in the temperature estimates as compared to the temperature reported by on-board sensors. Again, we assume that the temperature sensed by each sensor represents the true temperature of the core associated to it. We close with a note that the details of how the power is distributed inside the processor depends on the application being executed, and is difficult to obtain with sufficient accuracy for any state-of-the-art processor.

2.3 A New Approach

Simulators such as Hotspot are designed to be generic in order to support widest possible range of applications, use cases and processor models. As noted earlier, a side effect of such generality is that the system designer is burdened with specifying all the necessary information related to the processor and the runtime environment.

As a specific example, such simulators do not assume any properties about the application set to be executed, and therefore require detailed power trace (or model) information at the micro-architectural granularity to be provided by the system designer. However, if one restricts the application set such that the applications are deterministic, then it is possible to construct an accurate and computationally efficient thermal simulator which exploits the deterministic property of the application set. The immediate benefit of such an approach is that fewer parameters are required to be supplied by the system designer.

In the current context, we say that an application is deterministic in the following sense:

- The total power consumed by a given application is constant over time, and is determined by the application itself; and
- The distribution of the total power between the micro-architectural elements in the core is also constant.

Of course, an application may never be truly deterministic, as the total power consumption and its distribution in the core will depend many factors, such as the input to the application, hardware architecture, hardware state (e.g., cache eviction), and even temperature of the core. We show that these conditions are satisfied to a high degree by embedded and signal processing applications. As a result, when such a deterministic application executes, it leads to predictable and consistent changes in the temperature of the processor. This enables the resulting changes in temperature to be computed with sufficient accuracy (i.e., $<5^oC$ of error in a dynamic range of 80^oC) by knowing just the application and its scheduling details. In other words, an embedded or a signal processing application has a *thermal fingerprint*, enabling quick and accurate estimation of temperature from the knowledge of only a few parameters.

It must be reiterated that avoiding thermal faults requires investigating several (possibly hundreds) of different design choices (i.e., how the given applications be mapped to the available cores, how to design schedules to meet performance objectives, to name a few), which requires computational efficiency. To this end, we say that a thermal simulator (or a model) is computationally efficient if it can compute several thousand new temperature data points in the order of seconds or less. In this chapter, we say that a thermal simulator (or a model) is accurate if the error in the estimated temperature traces is less than 5^oC in the dynamic range of 80^oC.

In summary, the problem to be solved in this chapter is:

> *Given a possibly heterogeneous chip multiprocessor system* S *and a set of applications* A*: estimate the temperature trace on all cores of* S *with sufficient speed and accuracy as required for design space exploration. The method should not depend on prior availability of power, layout, physical and thermal models of the hardware platform.*

In order to solve the problem, a technique to construct the thermal model of the processor based on the concept of thermal fingerprints is proposed. The proposed approach requires access to temperature sensors on-board the processor. All parameters necessary for constructing the thermal model are automatically extracted by running a sequence of calibration experiments on the processor of interest. This model is then combined with mapping and scheduling information for estimating the desired temperature traces.

Furthermore, the proposed thermal model:

- Does not depend on the prior knowledge of details about the hardware platform, or the knowledge of power traces. The proposed approach can also be applied in case the given multiprocessor is heterogeneous;
- Can model and evaluate thermal effects of various mapping policies in terms of peak temperature experienced by processor cores, as well as changes in temperature of the processor over time and across cores;
- Can correctly determine a temperature trace even when two applications running on a chip multiprocessor (CMP) consume the same total power but result in different power density in the processor; and
- Allows for fast and accurate temperature estimation to be reliably used in DSE loops.

2.4 Setup and Notations

We consider a chip multiprocessor P with a set C of cores. For simplicity and without any loss of generality, we restrict the processor to operate at a single clock speed. The cores may be heterogeneous, i.e., a given core may belong to one of the types given in the set $\mathsf{V} = \{GPU, FPU, RISC, ...\}$. A set of embedded applications A is available for execution on the processor P. We assume that a set S of temperature sensors, and a set W of power sensors are available on the chip. Again, for simplicity, we assume that the sensors are noise-free. The restriction on the processor clock speed and the sensor noise will be relaxed in the subsequent chapters. During the construction of the thermal model, on-chip sensors are sampled periodically with a period t_s. Therefore, the construction of the model and subsequent estimation of temperature traces is done for discrete time instants $t \in \mathbb{Z}^{\geq 0}$. The temperature trace sampled from a given sensor is denoted as a tuple $\tau = \langle \tau_0, \tau_1, \cdots \rangle \in \mathsf{T}$ where τ_i is the temperature at the time instant $t = i$. Likewise, the power trace sampled from a given power sensor is denoted as a tuple $\rho = \langle \rho_0, \rho_1, \cdots \rangle \in \pi$ where ρ_i is the total power at the time instant $t = i$. A given application $a \in \mathsf{A}$ may or may not execute at a given time instant t, which is denoted by the associated utilization trace, $u = \langle u_0, u_1, \cdots \rangle \in \mathsf{U}$, with $u_i \in \{0, 1\}$, and $u_i = 0$ indicates that the application does not execute at time instant $t = i$. A core executing an application at a given time instant is considered to be active at that instant, else, it is considered to be inactive. Usually, $u \in \mathsf{U}$ is generated from a scheduling algorithm (e.g., round-robin). A binding $b = \langle a \in \mathsf{A}, u \in \mathsf{U}, c \in \mathsf{C} \rangle \in \mathsf{B}$ indicates that an application a executes according to a utilization trace u on a core c. Given a binding b, the helper function $\mathbf{a} : \mathsf{B} \to \mathsf{A}$ returns the application, the function $\mathbf{c} : \mathsf{B} \to \mathsf{C}$ returns the core, and the function $\mathbf{v} : \mathsf{B} \to \mathsf{V}$ returns the core-type. The function $\nu : \mathsf{C} \to \mathsf{V}$ provides the core-type for a given core. The function $\mathbf{d} : \mathsf{C} \times \mathsf{C} \to \mathbb{R}^{\geq 0}$ computes the Euclidean distance between two cores with respect to the layout of the multicore processor.

We also define the severity of thermal cycles experienced by the chip multiprocessor. Thermal cycles are periodic changes in temperature experienced by a core c, when a given subset $\mathsf{A}' \subseteq \mathsf{A}$ of applications execute on it. Large and rapid variations in the temperature of a processor degrades its reliability. To this end, we suppose that the processor is designed to withstand a certain maximum number of such temperature cycles before it fails, see [KBSM06]. Based on this concept, we define a simple metric E that measures the severity of thermal cycles experienced by the processor as under:

$$\mathsf{E} = \sum_{c \in \mathsf{C}} \sigma(T_c) \tag{2.2}$$

$\sigma(T_c)$ denotes the standard deviation of the temperature traces on the core $c \in \mathsf{C}$. A binding with a smaller E is preferable.

2.5 Constructing the Thermal Model

This section describes the construction of the thermal model M of chip multiprocessor P, given a set of deterministic applications, A. We call this technique "application fingerprinting", and it is based on two assumptions:

1. **Linearity of the Thermal Model**: This assumption is based on two counts: (i) the state-of-the-art numerical simulator Hotspot models a processor as a mesh of resistors and capacitors, and it is known that any mesh consisting of such passive electrical components forms a

linear circuit[1], see [Zum08, HoMAECE07], and (ii) the experimental results indicate that a linear thermal model can estimate the temperature traces with sufficient precision.

2. **Deterministic Runtime Behavior of Applications**: We assume that a given application $a \in \mathsf{A}$ is deterministic, i.e., it causes a constant total power consumption in a core it executes on, and furthermore, the distribution of this total power between the core's micro-architectural units is also constant. As a result, the changes in temperature of the core due to the application a is also deterministic, i.e., the changes in temperature are similar irrespective of the inputs presented to the application. Furthermore, we assume that the observed thermal changes are simple enough to be captured with sufficient precision into a thermal model. In other words, we suppose that each application $a \in \mathsf{A}$ has a "thermal fingerprint", which can be captured into a thermal model.

The overall procedure consists of two phases: (i) a calibration (or application fingerprinting) phase resulting in the required thermal model M, and (ii) a validation phase which tests the accuracy of the temperature traces estimated using the model M.

2.5.1 Application Fingerprinting

The goal of this stage is to compute a thermal model M which is a set of individual models $m \in \mathsf{M}$. A model m is derived such that it is possible to estimate the temperature trace at a distance d from an active core of type v executing an application a with the binding b. In particular, a model m belongs to the family of autoregressive moving average (ARMA) class of models, and has the form $m = \frac{B(z)}{F(z)}$ where $B(z)$ and $F(z)$ are polynomials in z^{-1}, the discrete time delay operator, see [DL80]. The function $\mathbf{m} : \mathsf{A} \times \mathsf{V} \times \mathbb{R}^{\geq 0} \rightarrow \mathsf{M}$ provides a model $m \in \mathsf{M}$ to compute the change in temperature at the distance d due to a core executing a binding b with an application $\mathbf{a}(b)$, on the core type $\mathbf{v}(b)$.

Subsequently, the temperature trace on a core c due to exactly one binding b is estimated as:

$$\tau^c = \quad \mathbf{m}(\mathbf{a}(b), \mathbf{v}(b), \mathbf{d}(\mathbf{c}(b), c)) \quad \otimes \quad \mathbf{u}(b) \tag{2.3}$$

where $\otimes$ is the convolution operator. Notice that in contrast to the conventional approaches, power traces are not required in estimating the temperature trace, τ^c.

---

[1] Under the assumption that thermal resistance of silicon does not vary significantly with temperature.

The overall temperature due to a set of bindings $\mathsf{B}' \subseteq \mathsf{B}$ simultaneously executing on P is then given by a simple superposition:

$$\tau^c = \sum_{b \in \mathsf{B}'} \{\mathbf{m}(\mathbf{a}(b), \mathbf{v}(b), \mathbf{d}(\mathbf{c}(b), c)) \quad \otimes \quad \mathbf{u}(b)\} \tag{2.4}$$

Presented below are two claims that enable the calculation of accurate temperature traces, without requiring any knowledge of the power density distribution in a core, or its power trace. Data extracted from the Simplescalar/Wattch simulator is used to support the claims being made on the relationships between total power, temperature and power-densities, due to an application $\mathbf{a}(b)$.

2.5.1.1 Non-Unique Relationship between Total Power Trace and Temperature Trace

A given total power trace associated with an application a executing on a core c of the processor P does not automatically imply a unique temperature trace. Instead, the distribution of total power (or power density) in the core determines the net flow of heat between various parts of the core, and hence, the overall temperature trace. Multiple applications can have the same total power consumption, but different power density distributions, causing a different overall temperature trace as observed at the temperature sensor. An example was presented in section 2.2.2. Thus, a correct thermal model must not calculate temperature traces solely as a function of total power traces, but must also consider how the power is distributed inside the processor.

2.5.1.2 Unique Relationship between application and Relative Power Distribution

Given a binding b, a deterministic application $\mathbf{a}(b)$ that executes on the core $\mathbf{c}(b)$ consumes a constant total instantaneous power, $\rho_i^{\mathbf{a}(b),\mathbf{v}(b)} = \rho_j^{\mathbf{a}(b),\mathbf{v}(b)}, i \neq j$ at time instants i, j provided $u_i = u_j = 1$. Furthermore, the distribution of this total power within the core is determined by the application $\mathbf{a}(b)$ itself. We show that such an assumption is a reasonable abstraction. To this end, we suppose that a particular application, such as a 16-point FFT will run through the same sequence of steps, irrespective of the inputs. In case the input to this application varies, these sequence of steps are repeated, and the number of accesses to each microarchitectural unit in the core also scales appropriately. Consequently, the total energy consumed by the application $\mathbf{a}(b)$ scales, but the total power and its distribution remains unchanged.

This claim was validated using several benchmarks from the MiBench Embedded Systems benchmarks suite, see [GRE$^+$01b]. The results for selected benchmarks are presented in Table 2.1. These benchmarks were run with varying inputs, which is reflected in the number of instructions

S.No	Application	#Runs	ΔP_{max}	Instructions Executed
1	FFT	10	4%	$64,892$ $354,675$
2	I-JPEG	10	2.7%	6.2×10^6 119.6×10^6
3	Matrix-Multiplication	10	1.2%	$85,910$ 107.6×10^6
4	GSM-Encoder (“toast”)	10	0.8%	5.6×10^6 19.6×10^6

Table 2.1: Power distribution statistics of selected benchmarks.

executed (column 5, minimum number of instructions vs maximum number of instructions executed). The variation in instantaneous total power consumption of an application executing under varying inputs is minimal. For instance, the maximum variation for the FFT application was only 4%, with inputs ranging from single-digit values to six-digit values. Similar results are obtained for other benchmarks.

In addition, power consumption in each micro-architectural unit for each of these benchmarks was also evaluated after making suitable modifications to the Wattch simulator. The results for FFT and GSM-Encoder (“toast”) application are shown in Figure 2.4. The same conclusions apply for other applications. It can be seen from the figure that the mean power consumption in all micro-architectural units in the core remains almost constant even under significant input variations. Almost all the difference in any total power consumption can be attributed to the variation in power consumed by the clock. Statistical parameters such as mode, median and standard-deviation are also shown in Figure 2.4. It can be observed that these statistical parameters also remain relatively unchanged.

From the preceding discussion, the following conclusions can be drawn:

1. Given a binding b, the instantaneous total power consumption of an application $\mathbf{a}(b)$ executing on a core of type $\mathbf{v}(b)$ at time $t = i$ is determined by the application itself:

$$\begin{cases} \rho_i^{\mathbf{a}(b),\mathbf{v}(b)} = \rho_j^{\mathbf{a}(b),\mathbf{v}(b)} & : \quad u_i = u_j = 1 \\ 0 & : \quad u_i = 0 \end{cases} \tag{2.5}$$

2. The power trace ρ^b (note the absence of a subscript) of an application $\mathbf{a}(b)$ is completely determined by the associated binding b. Specifically:

$$\rho^b \quad = \quad \mathbf{k}(\mathbf{a}(b), \mathbf{v}(b)) \times \quad \mathbf{u}(b) \tag{2.6}$$

where $\mathbf{k}(\mathbf{a}(b), \mathbf{v}(b))$ is a scalar constant determined by the application a and the core type v; $\times$ is a multiplication operator, and $\mathbf{u}(b)$ is the utilization trace associated with the binding b.

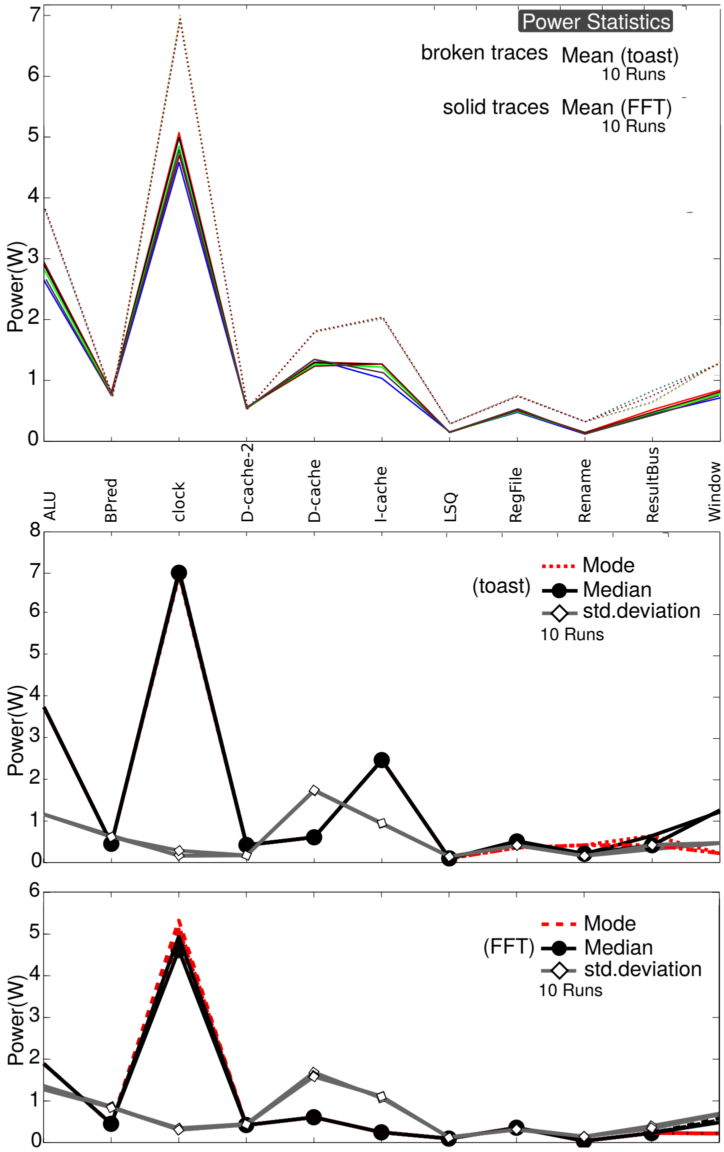


Figure 2.4: Statistics on power consumption for major micro-architectural units. Mean power consumption for 10 benchmarks (top), and main statistical parameters power consumption for the same runs (bottom).

3. Consider a binding b which executes on a core $\mathbf{c}(b)$. Assume that the associated power trace is ρ^b, and the corresponding temperature trace τ^c has been observed on a core $c \in \mathsf{C}$. Now, consider a thermal model $\overline{\mathbf{m}}(\mathbf{a}(b), \mathbf{v}(b), \mathbf{d}(\mathbf{c}(b), c))$ such that:

$$\tau^c = \rho^b \otimes \overline{\mathbf{m}}(\mathbf{a}(b), \mathbf{v}(b), \mathbf{d}(\mathbf{c}(b), c)) \tag{2.7}$$

   Now, consider another thermal model $\mathbf{m}(\mathbf{a}(b), \mathbf{v}(b), \mathbf{d}(\mathbf{c}(b), c))$ which only requires the knowledge of τ^c and the utilization trace $\mathbf{u}(b)$ such that the following relationship holds:

$$\tau^c = \mathbf{u}(b) \otimes \mathbf{m}(\mathbf{a}(b), \mathbf{v}(b), \mathbf{d}(\mathbf{c}(b), c)) \tag{2.8}$$

Requiring that the temperature traces calculated using either model be equal:

$$\begin{aligned} \mathbf{u}(b) \otimes \mathbf{m}(\mathbf{a}(b), \mathbf{v}(b), \mathbf{d}(\mathbf{c}(b), c)) &= \rho^b \otimes \overline{\mathbf{m}}(\mathbf{a}(b), \mathbf{v}(b), \mathbf{d}(\mathbf{c}(b), c)) \\ &= \{\mathbf{k}(\mathbf{a}(b), \mathbf{v}(b)) \ \times \mathbf{u}(b)\} \otimes \overline{\mathbf{m}}(\mathbf{a}(b), \mathbf{v}(b), \mathbf{d}(\mathbf{c}(b), c)) \\ &= \mathbf{u}(b) \otimes \{\mathbf{k}(\mathbf{a}(b), \mathbf{v}(b)) \times \overline{\mathbf{m}}(\mathbf{a}(b), \mathbf{v}(b), \mathbf{d}(\mathbf{c}(b), c))\} \end{aligned} \tag{2.9}$$

Therefore, we conclude:

$$\mathbf{m}(\mathbf{a}(b), \mathbf{v}(b), \mathbf{d}(\mathbf{c}(b), c)) = \mathbf{k}(\mathbf{a}(b), \mathbf{v}(b)) \times \overline{\mathbf{m}}(\mathbf{a}(b), \mathbf{v}(b), \mathbf{d}(\mathbf{c}(b), c) \tag{2.10}$$

In addition, we find:

$$\begin{aligned} \tau^c &= \rho^b \otimes \overline{\mathbf{m}}(\mathbf{a}(b), \mathbf{v}(b), \mathbf{d}(\mathbf{c}(b), c)) \\ &= \mathbf{u}(b) \otimes \{\mathbf{k}(\mathbf{a}(b), \mathbf{v}(b)) \times \overline{\mathbf{m}}(\mathbf{a}(b), \mathbf{v}(b), \mathbf{d}(\mathbf{c}(b), c))\} \end{aligned} \tag{2.11}$$

In other words, (2.6) through (2.11) show that the thermal model $\mathbf{m}(\mathbf{a}(b), \mathbf{v}(b), \mathbf{d}(\mathbf{c}(b), c))$ constructed using the temperature trace τ^c and the utilization trace, $\mathbf{u}(b)$ can be used to estimate correct temperature traces, without requiring any power trace information.

We conclude the discussion with the following note: given a binding $b = \langle a, u, c \rangle$ (where a is a deterministic application) it is possible to directly compute the temperature trace τ^c for any core c of the processor. However, the reverse may not be possible, i.e., given a temperature trace τ^c observed on the core c, it may not be possible to deterministically determine the binding b.

2.5.1.3 Constructing the Thermal Model

Figure 2.5 shows the overview of the application fingerprinting technique. The procedure starts with the design of a special utilization trace u^* together with an optimal set of calibration experiments.

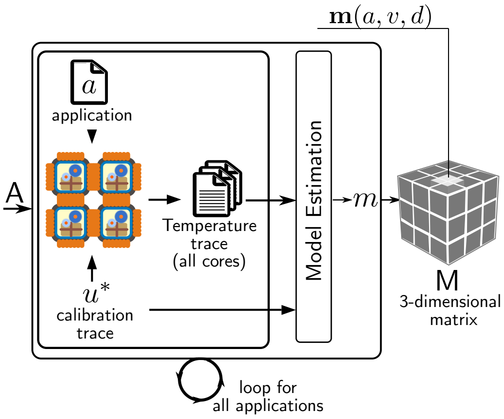


Figure 2.5: Construction of the thermal model M.

Calibration Experiments

We exploit the thermal fingerprint property of embedded and signal processing applications. The purpose of the calibration experiment is to capture this thermal fingerprint as a thermal model of each embedded application. Each experiment measures the change in temperature on the set $\mathsf{C}' \subseteq \mathsf{C}$ of cores caused by exactly one application $a \in \mathsf{A}$ executing on the processor P with the associated binding b. Thus, a calibration experiment is determined by (i) the application a, (ii) the associated binding b, and (iii) the set of cores $\mathsf{C}' \subseteq \mathsf{C}$ from which the temperature traces are recorded. The observed changes are then captured into a thermal model $m \in \mathsf{M}$. In principle, a separate experiment is required for each unique combination of application, core-type, and distance. In practice, the total number of calibration experiments may be less as the influence of a hot core on the temperature of other cores reduces significantly with distance due to high silicon thermal resistivity.

The Special Utilization Trace u^*

The purpose of this trace is to execute an application $a \in \mathsf{A}$ in a manner such that both the transient and steady-state thermal characteristics of the application can be accurately captured. Thus, the utilization trace consists of at least two distinct segments:

1. *Dynamics Segment*. This section of the calibration trace rapidly switches on and off the application (i.e., schedules the application in and out rapidly) in order to accurately capture the resulting transient changes in temperature; and

2. *Statics Segment*. This section of the calibration trace executes the application uninterrupted until all cores attain a steady state temperature. The application is then switched off until all cores again

reach a steady state temperature. This part of the calibration trace is designed to capture the steady state temperature change due to the application a.

The details are presented in Algorithm 1, in which an empty binding b is initialized, see line 1. Next, for every core type $v \in \mathsf{V}$, a *host core* $c^* = \mathbf{c}(b)$ is chosen to execute the application $\mathbf{a}(b)$ such that the resulting change in temperature over different distances can be observed in a single experiment, see line 4. The calibration experiment is performed for every unique core type $v \in \mathsf{V}$, and application $a \in \mathsf{A}$, see lines 2, 5, and 9. If there are multiples cores at a given distance d from $\mathbf{c}(a)$, then the average of the observations are taken, see line 19. This is done to mitigate errors due to asymmetrically located temperature hotspots. In other words, two temperature sensors at an equidistant location from a given hot core may sense slightly different temperature values as the hot core may not heat symmetrically with respect to the sensors. Thereafter, a model is estimated from the given calibration trace and the observed temperature trace (line 20), see [DL80].

The complexity of the assumed model may be varied by changing the number of poles (nf), the number of zeros (nb), and the initial discrete time delay (nk). The search for the best fitting model is an iterative procedure, gradually increasing the complexity of the model, see lines 26 - 41. The system designer may choose to restrict the maximum complexity of the temperature models by restricting the maximum number of poles, zeros and delay to `MAXPOLES`, `MAXZEROS` and `MAXDELAY` respectively. Between iterations, a new model is judged better than a previously computed one if the fit value of the new model exceeds that of the old one, see line 31.

It must be re-iterated that the proposed calibration based method is different from the traditional power-trace based approaches for constructing the thermal model of the processor, see [LTT08]. Instead, the proposed approach exploits the deterministic runtime behavior of embedded and signal processing applications to compute a thermal model from observing temperature changes as the given processor executes carefully designed bindings during the calibration phase.

2.5.2 Estimating Temperature Traces using the Thermal Model

The linearity property of the thermal model of M allows us to use the superposition principle for determining the overall temperature trace due a given set of bindings, $\mathsf{B}' \subseteq B$. The procedure for calculating detailed temperature traces is given in Algorithm 2.

The estimation procedure requires the set of bindings B' together with the associated thermal model M. Each binding $b \in \mathsf{B}'$ is evaluated to estimate the changes in the temperature of *all* cores $c \in \mathsf{C}$, see line 2. The changes in the temperature of a core c due to bindings in B' are superposed (i.e., element-wise addition of traces) in order to compute the final temperature trace.

```
Input: Applications A, Processor P
Output: Thermal Model M
Data: $t_s \leftarrow$ Discrete Sampling Interval;
Data: $T_{c,obs}, T_{d,obs}, d_{max} \leftarrow 0, d, H \leftarrow 0, u^*$;            // local variables
$b \leftarrow \langle \phi, \phi, \phi, \phi \rangle$                                  // Initial binding, all empty
foreach Core type $v \in$ V do
    $d_{max} \leftarrow 0, H \leftarrow 0$ ;
    Choose host core $\mathbf{c}(b) = c^* \mid \mathbf{d}(c^*, c_i) \geq \mathbf{d}(c_j, c_k), \nu(c^*) = v; c_i, c_j, c_k \in$ C ;
    foreach application $a \in$ A do
        Design the calibration trace $u^* \in$ U ;
        $T_{d,obs} \leftarrow 0, \forall d$ ;
        Execute $a$ according to binding $b = \langle a, u^*, c^* \rangle$;
                                                            // $\mathbf{u}^*(b) = u^*, \mathbf{a}(b) = a$
        foreach core $c \in$ C do
            $d \leftarrow \mathbf{d}(c, c^*)$;
            $T_{c,obs} \leftarrow$ Observed temperature trace from core $c$ ;
            $T_{d,obs} \leftarrow T_{d,obs} + T_{c,obs}$ ;
            $H[d] \leftarrow H[d] + 1$;                     // # cores at distance $d$ from $c^*$.
            if $d > d_{max}$ then
                $d_{max} \leftarrow d$ ;
            end
        end
        for $d = 0 : d_{max}$ do
            $T_{d,obs} \leftarrow T_{d,obs}/H[d]$;            // Mean change at distance $d$
            $\mathbf{m}(a, v, d)$ =EstimateModel ($T_{d,obs}, u^*, t_s$);
                                        // Store the model;
        end
    end
end

Function EstimateModel($\hat{T}, u^*, t_s$)
                    // $\hat{T}$: Observed temperature trace. $\hat{U}$: Calibration trace
    Data: fit $\leftarrow -1$, fit$' \leftarrow -1$;
                            // fitness of the estimated model, Maximum: 100%.
    Data: [nb, nf, nk] $\leftarrow$ [2, 2, 1] ;
                            // Initial order of model as vector [nb, nf, nk]
                                    // nb-1: number of zeros in the model
                                      // nf: number of poles in the model
                            // nk: discrete time delay (number of samples)
    Data: System Constraint: MAXPOLES, MAXZEROS, MAXDELAY;
    Data: $m$: Computed temperature model of type $\frac{B(z)}{F(z)}$ ;
    for nk = 1:MAXDELAY do
        for nb = 1:MAXZEROS do
            for nf = 1:MAXPOLES do
                Compute model $m' = \frac{B(z)}{F(z)} \mid u^* \otimes m' + e \approx \hat{T}$          // see [DL80]
                where e is the assumed error model;
                                                    // e.g., zero mean white noise
                if $fit > fit'$ then
                    fit $\leftarrow$ ComputeFit($\hat{T}$, $u^* \otimes m'$);
                    fit$' \leftarrow fit$ ;
                    $m \leftarrow m'$ ;
                                                        // the best model so far.
                    if $fit = 100$ then
                        return $m$;                             // perfect model.
                    end
                end
            end
        end
    end
    return $m$;

Function ComputeFit($\hat{T}, \hat{T}^*$)
    return $100 \times \left\{ 1 - \frac{\|\hat{T}^* - \hat{T}\|}{\|\hat{T} - \overline{\hat{T}}\|} \right\}$ ;
        // Normalized Root Mean Square Error estimate. $\overline{\hat{T}}$: Mean value of $\hat{T}$.
```

Algorithm 1: Calibration based algorithm to compute the temperature model M.

```
Input: Bindings $\mathsf{B}'$, Thermal Model $\mathsf{M}$
Output: Estimated Temperature Trace $\tau^c$ for $c \in \mathsf{C}$
Data: $t_s \leftarrow$ Discrete Sampling Interval;
Data: $\forall c \in \mathsf{C} : \tau^c = \oslash$                    // Initalize all traces to empty.
foreach $b \in \mathsf{B}'$ do
    // Iterate over all bindings
    foreach $c \in \mathsf{C}$ do
        $\tau^c = \tau^c + \{\mathbf{u}(b) \otimes \mathbf{m}(\mathbf{a}(b), \mathbf{v}(b), \mathbf{d}(\mathbf{c}(b), c))\}$ ;  // Temperature trace due to this binding on core $c$
    end
end
```

Algorithm 2: Estimating Temperature Traces from a set of bindings B'

2.6 Temperature Aware Design Space Exploration

Once the thermal model M is available, design space exploration can be performed to evaluate the effect of various bindings on the temperature profile over the chip multiprocessor, P. Currently available DSE tools can assist the system designer investigate various bindings, subject to a set of constraints and objectives, see [TCGK02]. These tools usually require the knowledge of the set of applications to be executed on the processor, and the knowledge of abstract hardware properties (number of cores, type of cores etc, but not the detailed floorplan) to be specified by the designer. One may integrate either a thermal simulator (e.g., Hotspot) or a lumped model into the selected DSE tool for thermal aware Design Space Exploration. As already pointed out, temperature estimation using a simulator requires the knowledge of hard to acquire parameters, whereas lumped models may not be sufficiently accurate. Instead, the thermal model constructed according to the propose approach can be easily integrated into the current tools to achieve the same objectives. A use case of a thermally aware DSE is shown in Figure 2.6. The DSE tool accepts the following parameters:

1. Abstract architectural properties: available computing resources, their types etc;
2. Set of binding constraints and objectives;
3. Set of applications, A; and
4. Feedback from the temperature evaluation framework which evaluates each binding for temperature characteristics (e.g., peak value, thermal cycles).

A candidate binding generated by the DSE tool is evaluated using the temperature evaluation component. Based on the feedback of the temperature evaluation component, the DSE tool may modify its internal parameters to rule out combinations that lead to unacceptable temperature profiles on P. Or, the DSE tool may successively refine bindings that are deemed to be favorable in terms of temperature. A simple approach used in successive refinement of bindings is simulated annealing. In this approach, starting from an initial binding b, applications are moved between cores, and temperature traces are re-calculated. The DSE tool also evaluates different scheduling policies for each core. The process continues till the required thermal constraints are satisfied (e.g., reduction in peak temperature and/or thermal cycles below a specified threshold).

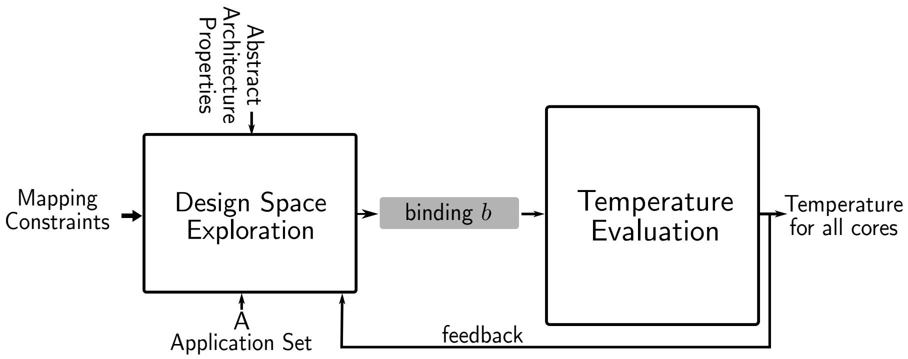


Figure 2.6: Temperature aware DSE Loop.

2.6.1 Sources of Inaccuracies

2.6.1.1 Inexact Thermal Model

Estimation of a thermal model m from a calibration trace and an associated temperature trace measurement is often an approximate process. Further, the order of the thermal model of m is limited to avoid dealing with overly complex impulse responses, thereby saving some computational effort. In this work, the accuracy of the thermal model m is specified with Normalized Root Mean Square Error (or simply *fit*) defined as:

$$fit_m = 100\left[1 - \frac{||\mathcal{T}^{*c} - \mathcal{T}^{c}||_2}{||\mathcal{T}^{*c} - \overline{\mathcal{T}^{*c}}||_2}\right] \qquad [\%] \tag{2.12}$$

Where:

$$\begin{array}{rcl} ||x||_2 & : & \text{The } l_2 \text{ norm of the vector } x \\ \mathcal{T}^{c} & : & \text{Temperature trace on core } c \text{ estimated using the model} \\ \mathcal{T}^{*c} & : & \text{Temperature trace observed on core } c \\ \overline{\mathcal{T}^{*c}} & : & \text{Mean value of temperature trace observed on core } c \end{array} \tag{2.13}$$

A *fit* of 100% indicates a perfect model, and depends on the application, distance from the hot core at which the temperature is to be estimated, the type of the hot core, and also the utilization trace. Since the maximum complexity of the model has been capped, the accuracy of the model is relatively less when the utilization trace may result in temperature transients. The *fit* reported in the experiments section is the worst calculated over several randomly generated utilization traces, ranging from

$u = [1, 1...1]$ (application is always executing) to $u = [1, 0, 1, 0...]$ (start-stop execution of application at time interval of t_s).

2.6.1.2 Under-estimating the impact of a hot core on a distant neighbor

It can be shown analytically, as well as from results published in recent literature that the thermal impact of a hot core drops rapidly with the distance from the thermal hotspot, see [WH11]. This is attributed to high lateral thermal resistance of silicon. The observations made during the calibration experiments can be used to estimate the distance d from the hot core beyond which one may choose to ignore the effect of heat transfer from the hot core.

One approach to choose d is described in the Algorithm 3. The distance d depends both on the application and the core-type, and therefore, the Algorithm investigates all core-types in the set V and all the applications in the set A. The input to the algorithm is the processor P, the application set A, as well as a parameter K chosen by the system designer. The meaning of the parameter K can be explained as follows:

1. Suppose that a hot core c^* executing an application has a thermal dynamic range δ^{c^*};
2. Let δ^c represent the thermal dynamic range observed on a core $c \in \mathsf{C}$ due to the hot core c^*; and
3. Then K selects the subset of cores $\mathsf{C}' \subseteq \mathsf{C}$ on which the following condition is satisfied: $\frac{\delta^{c^*}}{\delta^{c'}} \geq K$. In other words, $\mathsf{C}' \subseteq \mathsf{C}$ represents the set of cores on which the observed thermal effect of a hot core c^* has attenuated by a factor of at least K.

In order to achieve maximum thermal dynamic range for each application, a special utilization trace u^* is designed such that the processor can reach a steady state temperature when it executes an application under test, see line 1. Next, for each core-type, a core c^* belonging to the same type, and located as close to the center of the processor is selected, see line 4. Next, each application $a \in \mathsf{A}$ is executed on this core, and temperature traces from all cores are collected. Specifically, τ^c represents the temperature trace from core c, see line 6. The thermal dynamic range δ^{c^*} observed on the host core c^* is then evaluated, see line 7. We now find the set of cores $\mathsf{C}' \subseteq \mathsf{C}$ such that the observed thermal dynamic range $\delta^{c'}$ for each core $c' \in \mathsf{C}'$ satisfies $\frac{\delta^{c^*}}{\delta^{c'}} \geq K$, see line 8. For the considered core-type and application, the distances $\mathbf{d}(c^*, c') \mid c' \in \mathsf{C}'$ are computed, and a minimum of these distances d' is then stored, see line 10. At each evaluation, the information about the hottest application a^* encountered thus far, and the corresponding core type is recorded, see lines 13 and 14. The algorithm repeats for each application $a \in \mathsf{A}$ and core-type $v \in \mathsf{V}$ to yield the final value of d^{out}, the sought distance. The algorithm also provides a^{out} and c^{out} to be used in computing the related worst case error, see Algorithm 4.

```
Input: Application Set A, Processor P, Parameter $K$
Output: $d$: Distance beyond which the effect of a hot core may be ignored
Output: $c^{out}$: The host core to be used in Algorithm 4
Output: $a^{out}$: The application to be used in Algorithm 4
Data: $\forall c \in \mathsf{C} : \tau^c \leftarrow \oslash, d' \leftarrow 0, d \leftarrow \infty, \delta^{old} \leftarrow 0$    // Variables
Construct the utilization trace $u^* = [1, 1, \ldots]$;    // Long enough for the processor temperature to reach the steady state
foreach $v \in \mathsf{V}$ do
    // Iterate over all core types
    foreach $a \in \mathsf{A}$ do
        // Iterate over all core applications
        Choose a core $c^* \mid \mathbf{v}(c^*) = v$ and $c^*$ is located (approximately) at the center of P;
        Construct the binding $b = \langle a, u^*, c^* \rangle$ and execute on P;
        $\forall c \in \mathsf{C}, \tau^c$: temperature trace for core $c$;    // Observe temperature traces
        $\delta^{c^*} \leftarrow \mathbf{max}(\tau^{c^*}) - \tau_0^{c^*}$;    // $\delta^{c^*}$: thermal dynamic range observed on $c^*$
        $\mathsf{C}' \leftarrow \{c \in \mathsf{C} \mid \frac{\delta^{c^*}}{\delta^c} \geq K\}$;    // $\delta^c$: thermal dynamic range observed on $c$
        $\mathsf{D} \leftarrow \{\mathbf{d}(c^*, c') \mid c' \in \mathsf{C}'\}$;    // the set of (positive) euclidean distances
        $d' \leftarrow \mathbf{min}(D)$;
    end
    if $\delta^{c^*} > \delta^{old}$ then
        $a^{out} \leftarrow a$;    // $a^{out}$ is the hottest application so far
        $c^{out} \leftarrow c^*$;    // $c^{out}$ is the corresponding core
        $d \leftarrow d'$; // Hottest application will also largest thermal influence on other cores
        $\delta^{old} \leftarrow \delta^{c^*}$;    // Update
    end
end
```

Algorithm 3: Estimation of the distance d

```
Input: Application Set A, Thermal Model M, Processor P
Input: Distance $d$, application $a^{out}$, and core $c^{out}$ from Algorithm 3
Output: Worst Case Error $E$ due to a Distance Limited Model M
Data: $\tau_\infty^{c^{out},obs} \leftarrow \oslash, \tau_\infty^{c^{out},mdodel} \leftarrow \oslash$    // Local variables
Data: $t_s$ : Discrete Sampling Interval
Data: $E \leftarrow 0$    // Initalize.
Construct the utilization trace $u^* = [1, 1, \ldots]$;    // Long enough for the processor temperature to reach the steady state
$\mathsf{C}^* \leftarrow \{c : \mathbf{d}(c, c^{out}) \geq d\}$ // set of cores at a distance $d$ or greater from $c^{out}$;
$\mathsf{B}^* \leftarrow \{\langle a^{out}, c^*, u^* \rangle \mid c^* \in \mathsf{C}^*\}$;
$\tau_\infty^{c^{out},obs} \leftarrow$ Observed steady state temperature on the core $c^{out}$ when bindings $\mathsf{B}^*$ are executed on P
$\tau_\infty^{c^{out},model} \leftarrow$ Estimated steady state temperature on the core $c^{out}$ when bindings $\mathsf{B}^*$ are executed on P
$E = |\tau_\infty^{c^{out},obs} - \tau_\infty^{c^{out},model}|$
```

Algorithm 4: Worst Case Error Estimate

Ignoring the thermal influence of a hot core beyond a distance d reduces the computational effort for the computation of temperature traces, but at the cost of accuracy. In this case, the maximum possible error that can be incurred in temperature calculations must be determined. Assuming that we would like to ignore the temperature affects due to an active core beyond a distance $d > 0$, the resulting worst case error in the temperature estimates due to such an assumption must be calculated as shown in

Algorithm 4. The algorithm accepts a^{out} and c^{out} generated by Algorithm 3. The overall idea is to estimate the steady state temperature on core c^{out} when the processor executes a^{out} on all cores which are at a distance d or greater from c^{out}, and evaluate the corresponding error from measurement. The result is the required worst case error. The algorithm is self-explanatory, and the details are presented in Algorithm 4.

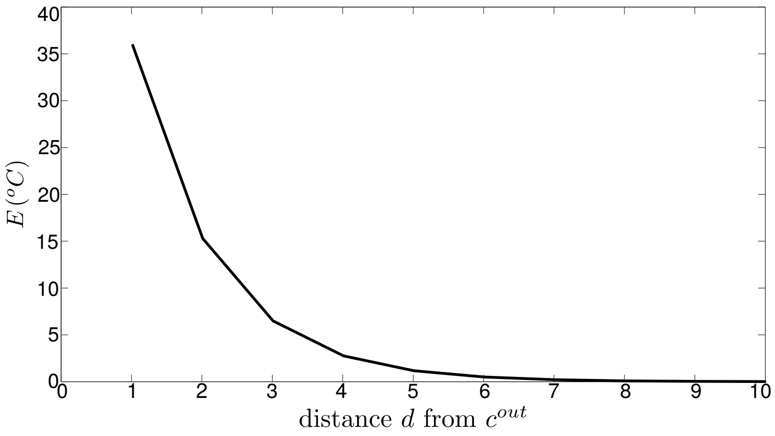


Figure 2.7: Maximum error in temperature estimate at c^{out} in a model M limited to distance d beyond which the system designer chooses to ignore the thermal effect of a hot core.

The values of E with respect to euclidean distance are shown in Figure 2.7. The plot was generated by varying the value of $1 \leq K \leq 8$ and extrapolating the results for $K > 8$. The application chosen was `susan`, executing on a 64-core Alpha 21264 processor supported by Hotspot, see Section 2.7 for further details. Though the error estimates presented in Figure 2.7 are specific to our experimental platform, the nature of the curve is expected to remain the same for any chip-multiprocessor platform. The results clearly show the risk associated with making any uncalculated simplifications on the impact of a hot core on its neighbors.

2.7 Experiments and Results

The approach presented in this chapter was validated using Hotspot. For this purpose, the specification of P was taken from the Magma project, which provides a variety of multicore floorplans consisting of 2 core- through 64 core- layouts, see [RV09]. Each core is an appropriately scaled version of the Alpha 21264 processor. The knowledge of physical arrangement of cores on the chip multiprocessor is required only if the user intends to apply approximations discussed in the Section 2.6.1.2. Such approximations were not made in our experiments. Although our technique does not require that all cores of P be homogeneous, the floorplans available in the Magma project consist of only homogeneous cores, and thus we

Application	0-hop	1-hop	2-hop	3-hop	> 3 hops
`splitstream`$\star$	99	99	95	89	83
`splitframe`$\star$	99	99	94	84	84
`iqzigzagidct`$\star$	99	99	92	89	83
`mergestream`$\star$	99	99	91	90	85
`mergeframe`$\star$	99	98	94	85	82
`Trigger`$\star$	99	98	91	91	86
`susan`†	99	99	95	88	84
`qsort`†	98	94	85	84	81
`toast`†	99	98	95	85	84
`untoast`†	99	99	96	90	87
`FFT`†	99	98	95	90	86
`bitcount`†	99	98	94	87	82
`basicmath`†	99	98	92	88	84
`adpcm`†	99	99	94	90	85
`LAME`†	99	99	95	87	83
`Matrix Multiplication`‡	99	98	90	89	83
`Producer`‡	98	98	92	87	80

Table 2.2: *Fit* of the estimated thermal model $m \in \mathsf{M}$. *: Motion-JPEG application split in 6 sub-applications[BBB^{+}11], †: MiBench Embedded Benchmark [GRE^{+}01b], ‡: Internal benchmark.

report results for a multiprocessor system with homogeneous cores. Furthermore, no power traces were used, neither in the estimation of impulse responses, nor in the calculation of any temperature traces. To demonstrate scalability of our technique, a large floorplan consisting of an 8x8 arrangement of cores was chosen. The following sections describe results relating to the *fit* of estimated impulse responses, as well as the accuracy of estimated temperature traces. Speedup due to our model as compared to Hotspot is also presented. The time resolution t_s is 1 ms.

2.7.1 Accuracy of Estimated Impulse Responses

The order of the thermal model was limited to 10, at which the *fit* achieved was greater than 90%. Further gains in *fit* with increase in the order of the model were insignificant ($< 0.1\%$). However, due to high lateral thermal resistance, the absolute error in temperature estimates small (5^{o}C). The results are summarized in Table 2.2. Only the worst *fit* per hop is reported for summary.

2.7.2 Speedup

A total of sixty bindings were evaluated, using applications from Table 2.2. The scheduling policy used on each core was varied between Earliest Deadline First (EDF), Least Laxity First (LLF), Rate Monotonic (RM), and Round Robin (RR). For each binding, temperature traces were calculated using our model, as well as Hotspot. The average time taken for such calculations using our model was 24.9 seconds, whereas Hotspot averaged about 6 hours. The summary is provided in Table 2.3. Further speedup is expected upon porting our algorithms to C/C++ from current Matlab/Java based environment.

Parameter(↓)	Our Model	Hotspot	Speedup
Mean Time (s)	24.853	29517	1187x
Maximum Time (s)	24.984	30057	1203x
Minimum Time (s)	24.795	27525	1110x
Standard Deviation (s)	0.054	547	

Table 2.3: Speedup achieved using the proposed approach as compared to Hotspot.

2.7.3 Accuracy of Estimation

We consider a special binding in which all applications are bound to cores located in a close spatial neighborhood. The temperature of each active core in this neighborhood is significantly influenced by all other active cores. Further, all active cores in this binding are scheduled according to round-robin (1-ms quantum) policy, where possible, leading to significant number of context switches, and thus causing rapid variations in temperature over time. Such a binding provides a good test for demonstrating the accuracy of temperature traces estimated using our technique.

The binding is shown in figure 2.8. Taking the core with <5,5> as the center, applications are mapped on the immediate 1-hop neighborhood, totaling 9 heat generating cores. All other cores in this case are idle. The results are shown is shown in Figures 2.9. It is clear that the binding in Figure 2.8 leads to significant thermal cycles on almost all active cores. Furthermore, our thermal model was able to calculate correct temperature traces for all cores, well within the accuracy goal set up in the introductory section of this chapter, see Figure 2.11.

In section 2.2.2, the applications *producer* and *FFT* consume the same total power, but differ in their respective power density distributions. Both these applications were mapped onto core <4,4>, see Figure 2.8. It can be seen from temperature trace for core <4,4> in Figure 2.9 that producer and FFT applications produce distinctly different temperature affects. Our model was able to accurately capture the effect of differences in power density distribution between *producer* and *FFT*. Figure 2.10 shows a section of temperature trace for core <4,4> from Figure 2.9 for more clarity.

Other bindings in which active cores are not immediate neighbors were also evaluated, and the prediction error was lower than what is reported in Figure 2.11. This is because an active core was not significantly influenced by its neighbors. Under these circumstances, estimation errors due to relatively lower *fit* were limited by high thermal resistance, see Table 2.2.

A thermal aware DSE use-case may be to reduce the thermal cycles experienced by the processor, given a set of applications to be executed, and

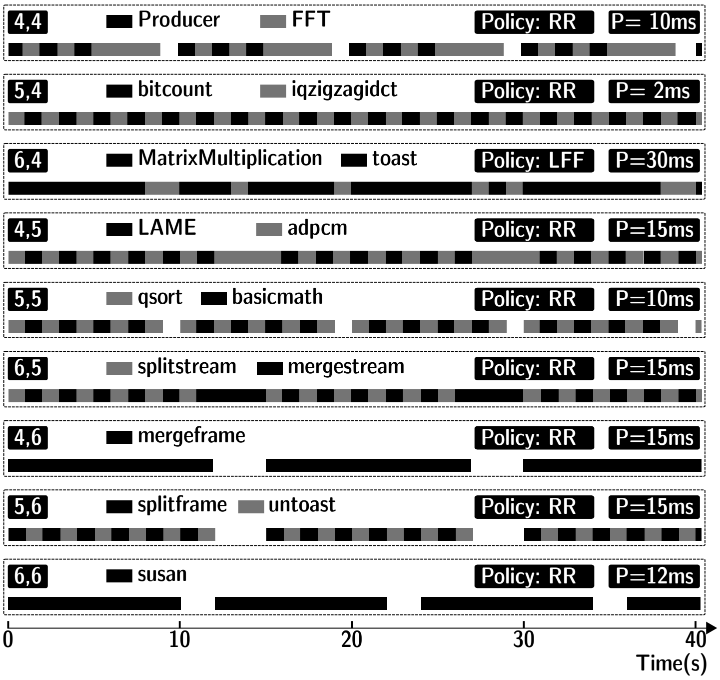


Figure 2.8: Schedule for nine cores. Time scale (ms) is indicated. Letter 'P' shows the period of each schedule. Scheduling policy for each core is also indicated.

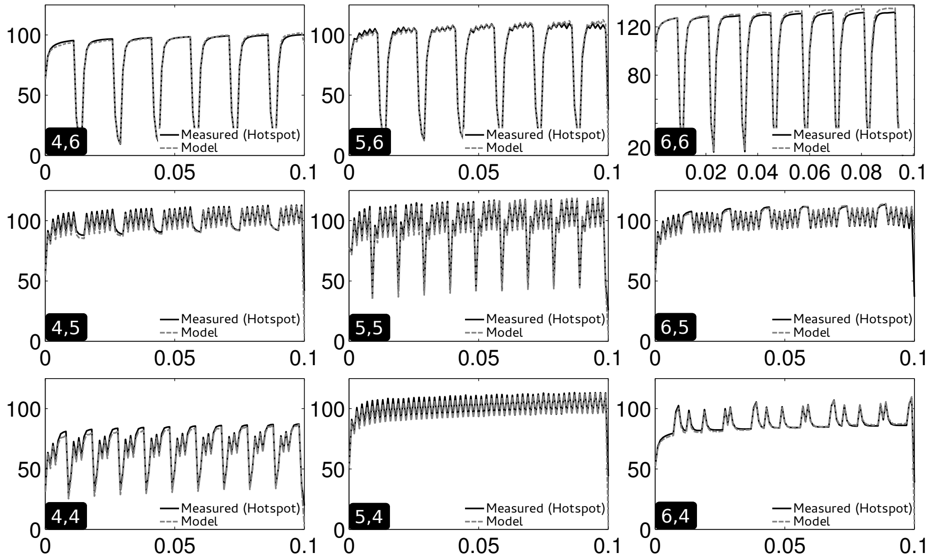


Figure 2.9: Measured temperature vs estimated temperature for nine active cores. Horizontal axes is time in seconds, vertical axes is $\Delta T(^oC)$. E $= 47.7$.

their respective schedules. Specifically, a feasible solution must ensure that all applications are schedulable, whereas the thermal cycles are minimized. In this case, the DSE tool may investigate several application-to-core mapping options till a suitable design solution is found, i.e., a new

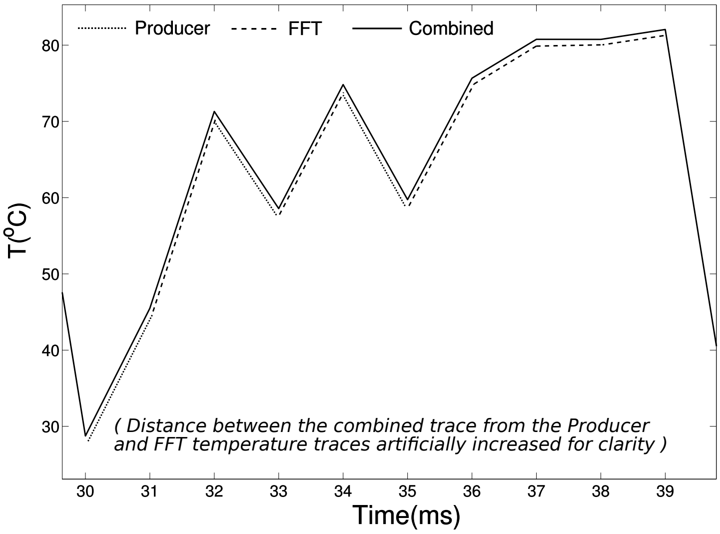


Figure 2.10 Section of temperature trace on core <4,4>. Both FFT and Producer applications have the same total power consumption, but lead to different temperature traces.

solution which results in a lower E is admitted if all applications can be executed as per the provided schedules. An example is presented in Figure 2.12. Notice that the total work done by each application remains unchanged. For instance, LAME executes for a total of 6ms, with a period of 15ms in both bindings. Also, a scheduling policy which minimized the number of context switches was chosen. See Figures 2.12 and 2.13. The overall error in estimated temperature is similar to the result shown in Figure 2.11; with the maximum error being 4.7^oC. It is not always possible to reduce thermal cycles by changing the bindings of applications, for a feasible scheduling policy for all cores may not exist. When estimating temperature traces, the thermal influence of each hot core was evaluated across the entire platform (i.e., the distance $d \leftarrow \infty$, see Section 2.6.1.2).

2.8 Variations and Optimizations

It may be possible to reduce the number of thermal models to be estimated if the given systems has thermal symmetry. In this case, all cores are first classified into a set of thermally different locations (TDLs), see [CM10]. During the calibration step, applications need to be executed on only one distinguished core in each TDL. In the estimation stage, the calculation of thermal effect of an active core c on core c' proceeds in two steps. First, a sequence of transformations is determined which translates c to a core in one of the TDLs. Next, the same sequence of transformations is applied to core c'. This preserves the relative location of cores c and c'. Temperature trace calculation can now proceed normally. Thermal symmetry reduces the memory space required to compute and store the thermal model M.

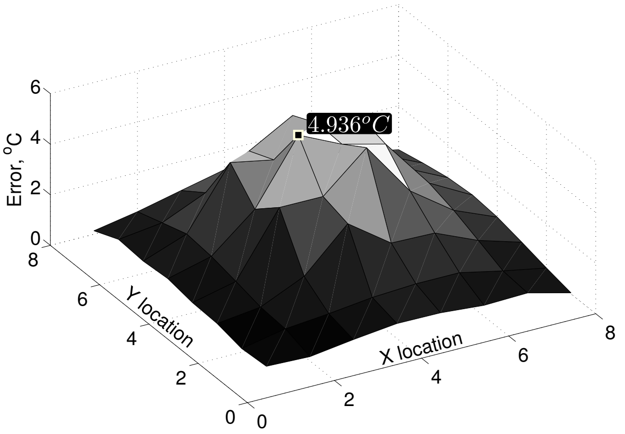


Figure 2.11 Maximum error in prediction over entire P. Errors in absolute values.

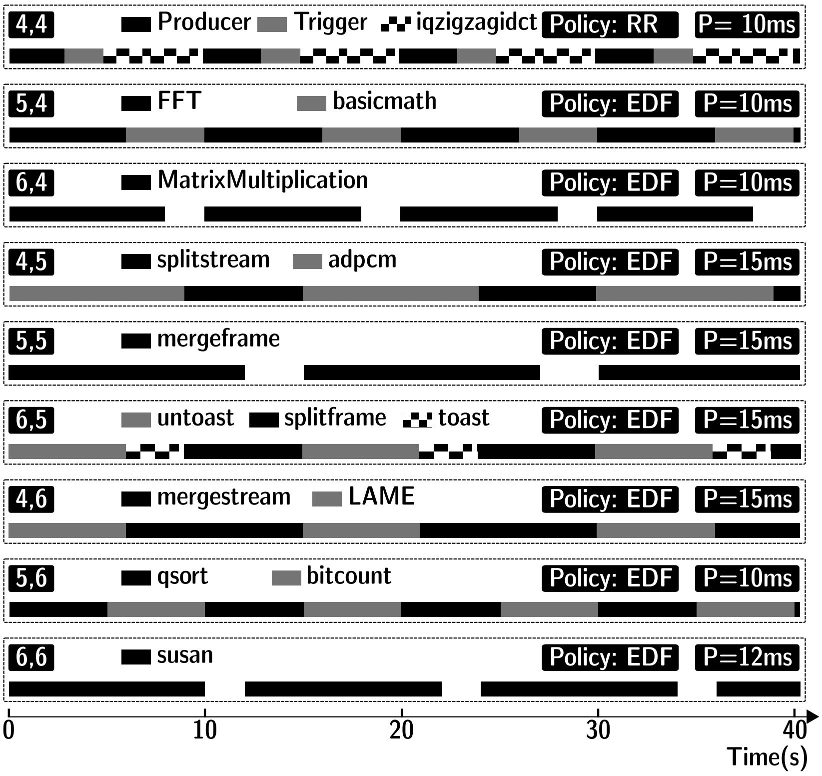


Figure 2.12 New binding derived from the one in Figure 2.8.

The computational load for estimating the temperature trace for a given binding may not change much, if large errors in temperature estimation are to be avoided, see section 2.6.1.2.

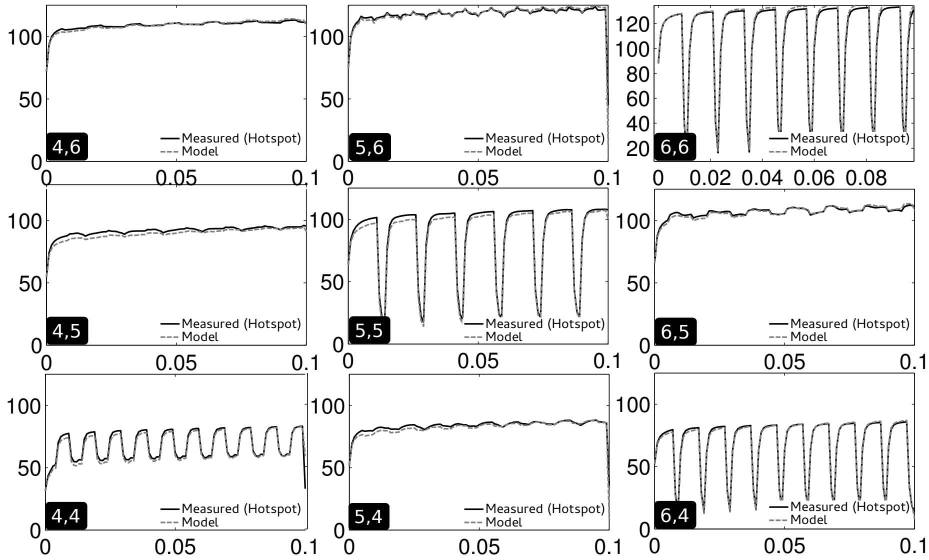


Figure 2.13 Reduction of thermal cycles by changing the bindings of applications. Horizontal axes is time in seconds, vertical axes is $\Delta T(^oC)$. E $= 25.1$

2.9 Closing Remarks

The chapter presented a new calibration based approach for constructing an accurate and computationally efficient thermal model of the given processor. Using a set of specifically designed calibration experiments and observations from on-board temperature sensors, the approach is able to extract all the parameters necessary for constructing the thermal model. The speed and accuracy of the proposed approach enables the exploration of a large set of candidate bindings using the design space exploration loop. The highlight of the proposed approach is that it does not require any power-trace information, or the hard-to-obtain details about hardware, such as the detailed floorplan. The proposed technique can also account for differences in power densities on a core due to an application, even when the total power consumed by two or more applications is the same. This makes our technique applicable to any given set of embedded applications and hardware.

Conventional power-trace based approaches require physical access to the processor in order to construct the thermal model. Often, this is in order to instrument the processor for accessing power traces and other necessary parameters about the processor. In contrast, the proposed technique does not require physical access to the processor, which can be exploited by system designers to construct and recalibrate system thermal models *in-situ*. It is known that physical properties of processors from the same family (and even from the same die) differ from each other due to small variations in the manufacturing process. The thermal characteristics of the processor are also influenced by its packaging, casing, and other environmental factors (e.g., field conditions). As a result, each computer system's thermal model is unique. It follows that if the system-level temperature management algorithms are to work effectively, then the thermal model of each individual system must accurately capture its thermal characteristics. The capability to construct and update the thermal model of each system purely by software means, without requiring physical access to the system can be exploited by system vendors to ensure that each system's temperature management algorithms are working as effectively as intended, irrespective of the difference in the overall thermal properties between individual systems. The system thermal model can be automatically, and even remotely upgraded if the system components are changed (e.g., new cooling system, new case etc) ensuring that the temperature control algorithms are always precise over the lifetime of the device.

3

Thermal Models for State-of-the-Art Processors

Summary

This chapter adapts the methods proposed in Chapter 2 for constructing thermal models for state-of-the-art processors. Furthermore, some restrictions from Chapter 2 are also relaxed, making the adapted methods applicable to processors which can operate at several clock speeds, featuring noisy temperature sensors, and processors which implement on-chip power and temperature management functions. The new methods are validated against the latest Intel Xeon 8-core processor. Extensive experimental results presented in this chapter suggest that the methods proposed in Chapter 2 are fundamentally sound, and the proposed scheme can be used to estimate temperature traces of state-of-the-art processors with reasonable accuracy, and with limited computational effort.

3.1 Introduction

The previous chapter presented a calibration based technique for constructing a thermal model of a given processor, without requiring the knowledge of hard-to-get system parameters such as the detailed processor floorplan. Succinctly, a sequence of carefully constructed calibration experiments are executed on the processor for which a thermal model is desired. The resulting temperature traces are then analyzed to extract the parameters necessary for the required thermal model.

The proposed approach was motivated by the challenges faced by a system designer in using the currently available thermal models. Specifically, high accuracy thermal simulators such as Hotspot require the designer to have an accurate knowledge of several system parameters, some of which may be difficult to acquire with sufficient precision, e.g., processor floorplan. On the other hand, lumped models abstract away lot of processor details (e.g., power distribution in the core) and therefore require relatively fewer parameters to be supplied. However, lumped models also tend to be less accurate as compared to numerical simulators, making them unsuited in certain use-cases, such as Design Space Exploration on state-of-the-art multicore processors. The approach proposed in Chapter 2 is able to overcome these challenges by extracting the parameters required for a thermal model by observing the temperature traces when the processor executes a sequence of carefully planned calibration experiments.

It must be pointed out that the resulting model cannot be compared directly to other solutions, such as the Hotspot numerical simulator. The Hotspot simulator is generic, and has been designed to support as many processor models, applications, and use-cases as possible. In contrast, the calibration based technique proposed in the previous chapter results in a *set* of thermal models, wherein each individual model is specific to a given application, processor clock speed, core-type, and of course, the processor model. Furthermore, the proposed methods in Chapter 2 for constructing the thermal model of a processor are restricted to embedded or signal processing applications which are deterministic in the following sense: (i) the total power consumed by a given application is determined by the application itself, and is also constant over time, and (ii) the distribution of the total power between the micro-architectural elements in the core is also constant. As argued in Chapter 2, it is understood that practical applications may not be strictly deterministic, since the total power consumption and its distribution in the core will depend on many factors, such as the input to the application, hardware architecture, hardware state (e.g., cache eviction), and even the temperature of the core on which the application executes. However, it was shown in Chapter 2 that the conditions above are satisfied to a high degree by embedded and signal processing applications. As a result, when such a deterministic application executes, it leads to predictable and consistent changes in the temperature of the processor, making is possible to estimate temperature traces from the knowledge of just the application and the corresponding scheduling information.

The previous chapter also made a few simplifying assumptions, such as the absence of noise in the temperature sensors, and restricting the processor to operate at a single clock speed. In addition, estimating the temperature trace on a given core when more than one core executes applications required only a simple superposition of traces, see Algorithm 2.

This chapter focuses on constructing thermal models for state-of-the-art processors. Accordingly, the algorithms proposed in Chapter 2 are adapted to allow:

- Noise in the temperature sensors on-board the processor;
- A set of discrete clock speeds at which the given processor can operate; and
- Power and temperature management algorithms which are common in state-of-the-art processors.

This chapter assumes that the processor is cooled at a constant rate, e.g., in the case of a fan, it is set to a constant speed. The next chapter will present a method to include the effect on the system fan into the thermal model.

3.2 Brief Problem Statement and Related Work

As noted earlier, the focus of this chapter is to adapt the methods presented in Chapter 2 for constructing a thermal model for a state-of-the-art processor. The overall challenge remains similar as before: we seek a method to construct the thermal model of the given processor which is not dependent on the system designer's knowledge of the properties of the system, and can instead extract the parameters necessary by observing the system of interest.

Specifically, the problem statement considered in this chapter can be summarized as:

Given a processor P *with a set of on-chip temperature sensors* S *and a set of embedded applications* A*, construct an accurate and fast thermal model* M *of the processor* P *without requiring access to any hard-to-get parameters such as detailed power trace(s) or the floorplan of the processor.*

The model M is said to be accurate if the error in the estimated traces always remains less or comparable to the noise in the on-chip temperature sensors on the processor P; and is said to be fast if it takes an order of seconds to compute few tens of thousands of data points.

The approach proposed in this chapter uses temperature sensors onboard the given state-of-the-art processor for constructing and validating the thermal model. It is assumed that the sensors are reliable and a readout from a given sensor can be taken as the representative temperature of the location where the sensor is placed. Specifically, in the case of a multicore system featuring one sensor per core, the readout from a given sensor is taken to be the true (or representative) temperature of the associated core. Consequently, if the temperature traces computed using our thermal model agree with the readouts from the sensors, then the validation is taken to be correct and complete. A direct comparison of our results against a corresponding Hotspot thermal model (which does not yet exist) is infeasible due to significant challenges involved in accurately constructing such a model, as already pointed out.

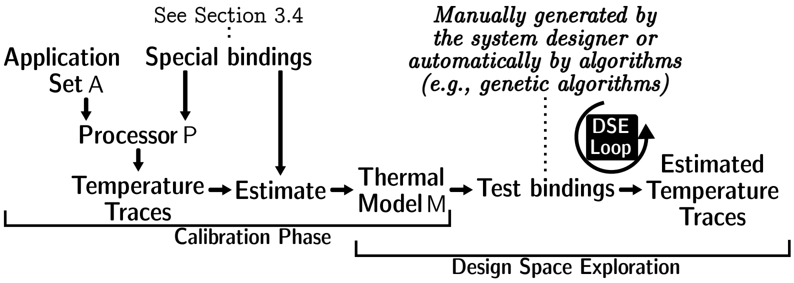


Figure 3.1: Overview of the approach.

Contribution

The chapter makes two contributions to the state-of-the-art:

1. Suitable adaptations to methods proposed in Chapter 2 in order to target state-of-the-art processors which can (i) operate at multiple clock speeds, (ii) feature power and temperature management algorithms implemented in hardware, and (iii) possess noisy on-chip temperature sensors;
2. Validation of the model by extensive experiments on a state-of-the-art multicore processor, i.e., the Intel Xeon 8-core processor.

3.3 Overview of the Approach

Construction of the thermal model requires (i) access to the multiprocessor P, (ii) the set of embedded applications A which will be executed on P, and (iii) a special set of bindings to be used for calibration experiments. See Figure 3.1.

During the calibration phase, the applications from the set A are executed as per specifically designed bindings on the processor and the resulting temperature traces collected. We exploit the deterministic property of embedded and signal processing applications, and estimate a transfer function between the application bindings and the observed *changes* in the temperature of the processor P, see Section 2.3 for the discussion on the assumed deterministic properties, and Section 3.4 for the definition of bindings. Thus, in contrast to current approaches wherein constructing a thermal model requires access to hard-to-get parameters, the proposed calibration experiments are designed to extract *all* the necessary parameters required by the thermal model. During the DSE phase, the system designer may explore *arbitrary* bindings of the applications in A until a solution satisfying the given set of temperature and performance constraints is discovered. It must be pointed out that though the same application set A is used during model construction (i.e., calibration) and DSE, the bindings used in these two phases may be completely unrelated.

3.4 Setup and Notation

We consider a chip multiprocessor P with a set C of cores. Each core can operate at set of discrete clock speeds in the set F. The cores may be heterogeneous, i.e., a given core may belong to one of the types given in the set $\mathsf{V} = \{GPU, FPU, RISC, ...\}$. A set of embedded applications A are available for execution on the processor P. During the construction of the thermal model, on-chip temperature sensors are sampled periodically with a period t_s. Therefore, the construction of the model and subsequent estimation of temperature traces is done for discrete time instants $t \in \mathbb{Z}^{\geq 0}$. The temperature trace sampled from a sensor is denoted as a tuple $\tau = \langle \tau_0, \tau_1, \cdots \rangle \in \mathsf{T}$ where τ_i is the temperature at the time instant $t = i$. A given application $a \in \mathsf{A}$ may or may not execute at a given time instant t, which is denoted by the associated utilization trace, $u = \langle u_0, u_1, \cdots \rangle \in \mathsf{U}$, with $u_i \in \{0, 1\}$, and $u_i = 0$ indicates that the application does not execute at time instant $t = i$. A core executing an application at a given time instant is considered to be active at that time instant, else, it is considered to be inactive. Usually, U is generated from a scheduling algorithm (e.g., round-robin). We assume that a set S of temperature sensors are available on the chip. A binding $b = \langle a \in \mathsf{A}, u \in \mathsf{U}, c \in \mathsf{C}, f \in \mathsf{F} \rangle \in \mathsf{B}$ indicates that an application a executes according to a utilization trace u on a core c which operates at a clock speed f. Given a binding b, the helper function $\mathbf{a} : \mathsf{B} \to \mathsf{A}$ returns the application, the function $\mathbf{c} : \mathsf{B} \to \mathsf{C}$ returns the core, the function $\mathbf{f} : \mathsf{B} \to \mathsf{F}$ returns the operating clock speed of the core, and the function $\mathbf{v} : \mathsf{B} \to \mathsf{V}$ returns the core-type. The function $\nu : \mathsf{C} \to \mathsf{V}$ provides the core-type for a given core. The i^{th} binding from the set B is denoted as b_i. The function $\mathbf{d} : \mathsf{C} \times \mathsf{C} \to \mathbb{R}^{\geq 0}$ computes the Euclidean distance between two cores with respect to the layout of the multicore processor.

3.5 Constructing the Thermal Model

Overview

The overall procedure consists of a calibration phase with two parts: (i) construction of a thermal model M and a function **g** which is used to compute the change in the temperature of a core c due to *exactly* one application $a \in \mathsf{A}$ executing on the processor P with the binding $b \in \mathsf{B}$, and (ii) construction of the function **h** which together with M and **g** is used to compute the overall change in the temperature of a core $c \in \mathsf{C}$ due to multiple applications executing on the processor, each with its own binding. The function **h** also captures the thermal effect of the power and/or temperature management algorithms implemented in the processor.

3.5.1 The Thermal Model M

The notations used in this section are similar to those used in the previous chapter, see Section 2.5.1, and are being repeated here for the convenience of the reader. The thermal model is a set M of individual models $m \in \mathsf{M}$. A model m is derived such that it is possible to estimate the temperature trace at the distance d from an active core of type v executing an application a with the binding b. In particular, $m \in \mathsf{M}$ is an output-error (OE) model from the autoregressive moving average (ARMA) family of models, and has the form $m = \frac{B(z)}{F(z)}$ where $B(z)$ and $F(z)$ are polynomials in z^{-1}, the discrete time delay operator, see [DL80]. The function $\mathbf{m} : \mathsf{A} \times \mathsf{V} \times \mathbb{R}^{\geq 0} \times \mathsf{F} \to \mathsf{M}$ provides a model $m \in \mathsf{M}$ to compute the change in temperature at the distance d due to a core executing a binding b with an application $\mathbf{a}(b)$, on the core type $\mathbf{v}(b)$, and at clock speed $\mathbf{f}(b)$. Notice that this chapter has relaxed one of the constraints in the previous chapter by explicitly allowing for the processor to operate at a discrete clock speed in the set F. The construction of the thermal model requires designing a special utilization trace u^* together with an optimal set of calibration experiments as follows:

The Calibration Experiments

We exploit the observation that a given embedded application exercises the processor in a well defined manner consistent across invocations and inputs, resulting in temperature changes on the processor which is uniquely related to the given embedded application itself, also referred to as the *thermal fingerprint*, see Chapter 2. Therefore, the purpose of the calibration experiment is to capture the thermal *impulse response* of each embedded application. Each experiment measures the change in temperature on the set $\mathsf{C}' \subseteq \mathsf{C}$ of cores caused by exactly one application $a \in \mathsf{A}$ executing on the processor P with the associated binding b. Thus, a calibration experiment is determined by (i) the application a, (ii) the associated binding b, and (iii) the set of cores $\mathsf{C}' \subseteq \mathsf{C}$ from which the temperature traces are recorded. The observed changes are then captured into a thermal model $m \in \mathsf{M}$. In principle, a separate experiment is required for each unique combination of application, core-type, distance and clock speed of the core. In practice, the total number of calibration experiments are less due to two factors:

1. The influence of a hot core on the temperature of other cores reduces significantly with distance due to high silicon thermal resistivity, which in the presence of sensor noise may deteriorate the signal-to-noise (SNR) to the extent that constructing a model may be infeasible; and

2. It may be possible to group applications based on similarities in their thermal impulse responses. However, we do not explore this possibility in the current thesis.

The Special Utilization Trace u^*

The purpose of this trace is to execute an application $a \in \mathsf{A}$ in a manner such that both the transient and steady-state thermal characteristics of the application can be accurately captured. This special utilization trace (also the *calibration trace*) consists of a dynamics segment, meant capture the transient thermal impact of an application, and a statics segment, meant to capture the steady-state thermal behavior of the same application. Details were presented in previous chapter, and therefore, we do not repeat them once again, see Section 2.5.1.3.

The details are presented in Algorithm 5. Note the similarity with Algorithm 1, except for an additional variable f indicating processor clock speed, see line 9. In order to avoid unnecessary repetition, we skip a detailed discussion on the mechanics of the algorithm. However, it must be reiterated that a model is separately computed for each discrete clock speed $f \in \mathsf{F}$ since the thermal properties of the system may change considerably with clock speed. In other words, it may not be possible to simply scale a given model derived at one clock speed and use it at a different clock speed. If this is not the case, an additional optimization step may be performed to remove redundant models.

The Function g

Given the thermal model M, the change in temperature on the core c due to *exactly one* binding b being executed on the processor can be computed using the function $\mathbf{g}$:

$$\tau = \mathbf{g}(b, \mathsf{M}, c) \overset{\text{def}}{=} \mathbf{m}(a, \mathbf{v}(b), \mathbf{d}(\mathbf{c}(b), c), \mathbf{f}(b)) \otimes \mathbf{u}(b) \tag{3.1}$$

Where $\otimes$ is the convolution operator and τ is the temperature trace. Notice the underlying assumption that the thermal model M is linear. Therefore, if a core dynamically switches between operating clock speed and tasks, separate temperature traces are computed for each task-clock speed configuration and added point-wise to yield the final temperature trace.

The Function h

The superposition function computes the final temperature trace for each core when multiple cores simultaneously execute a set of bindings B. Due to power and/or temperature management algorithms implemented in the

```
Input: Applications A, Processor P
Output: Thermal Model M
Data: $t_s \leftarrow$ Discrete Sampling Interval;
Data: $T_{c,raw}, T_{d,obs}, d_{max} \leftarrow 0, d, H \leftarrow 0, u^*$;    // local variables
$b \leftarrow \langle \phi, \phi, \phi, \phi \rangle$    // Initial binding, all empty
foreach Core type $v \in$ V do
    $d_{max} \leftarrow 0, H \leftarrow 0$ ;
    Choose host core $\mathbf{c}(b) = c^* \mid \mathbf{d}(c^*, c_i) > \mathbf{d}(c_j, c_k), \nu(c^*) = v; c_i, c_j, c_k \in$ C ;
    foreach application $a \in$ A do
        foreach clock speed $f \in$ F do
            Design the calibration trace $u^* \in$ U ;
            $T_{d,obs} \leftarrow 0, \forall d$ ;
            Execute $a$ according to binding $b = \langle a, u^*, c^*, f \rangle$;
                                                    // $\mathbf{u}^*(b) = u^*, \mathbf{a}(b) = a$
            foreach core $c \in$ C do
                $d \leftarrow \mathbf{d}(c, c^*)$;
                $T_{c,raw} \leftarrow$ Observed temperature trace from core $c$ ;
                $T_{d,obs} \leftarrow T_{d,obs} +$ Denoise ($T_{c,raw}$) ;
                $H[d] \leftarrow H[d] + 1$;    // # cores at distance $d$ from $c^*$.
                if $d > d_{max}$ then
                    $d_{max} \leftarrow d$ ;
                end
            end
            for $d = 0 : d_{max}$ do
                $T_{d,obs} \leftarrow T_{d,obs} / H[d]$;    // Mean change at distance $d$
                $\mathbf{m}(a, v, d, f) =$ EstimateModel ($T_{d,obs}$, $u^*$, $t_s$);
                                    // Store the model;
            end
        end
    end
end

Function EstimateModel($\hat{T}, u^*, t_s$)
                // $\hat{T}$: Observed temperature trace. $u^*$: Calibration trace
                                                    // $t_s$: Sampling interval
    Data: fit $\leftarrow -1$, fit$' \leftarrow -1$;
                        // fitness of the estimated model, Maximum: 100%.
    Data: [nb, nf, nk] $\leftarrow$ [2, 2, 1] ;
     // Initial order of model as vector [#zeros+1, #poles, #delay(samples)]
    Data: System Constraint: MAXPOLES, MAXZEROS, MAXDELAY;
    Data: $m$: Computed temperature model of type $\frac{B(z)}{F(z)}$ ;
    for nk = 1:MAXDELAY do
        for nb = 1:MAXZEROS do
            for nf = 1:MAXPOLES do
                Compute model $m' = \frac{B(z)}{F(z)} \mid t_s,\ u^* \otimes m' + e \approx \hat{T}$    // see [DL80]
                where e is the assumed error model;
                                            // e.g., zero mean white noise
                if $fit > fit'$ then
                    fit $\leftarrow$ ComputeFit($\hat{T}$, $u^* \otimes m' + e$);
                    fit$' \leftarrow fit$ ;
                    $m \leftarrow m'$ ;
                                                    // the best model so far.
                    if $fit = 100$ then
                        return $m$;                 // perfect model.
                    end
                end
            end
        end
    end
    return $m$;

Function ComputeFit($\hat{T}, \hat{T}^*$)
    return $100 \times \left\{ 1 - \frac{\|\hat{T}^* - \hat{T}\|}{\|\hat{T} - \overline{\hat{T}}\|} \right\}$ ;
      // Normalized Root Mean Square Error estimate. $\overline{\hat{T}}$: Mean value of $\hat{T}$.

Function Denoise($T_{raw}$)
     // Remove noise from the input trace using edge-preserving algorithms,
    e.g., Daubechies wavelets.
    return Denoised temperature trace;
```

Algorithm 5: Calibration based algorithm to compute the temperature model M. Similar to Algorithm 1, except for an additional variable f indicating processor clock speed.

processor, the final temperature trace τ for the core c may not be a simple superposition of temperature traces, also see Figure 3.5.

$$\tau \neq \sum_{b \in \mathsf{B}} \mathbf{g}(b, \mathsf{M}, c) \tag{3.2}$$

The exact thermal impact of such algorithms will vary between the processors. However, from the observations on the Intel Xeon processor, it is sufficient to use an application specific damping function $\mathbf{s} : \mathsf{A} \times \mathsf{V} \to \mathbb{R}^{>0}$ (details in the following paragraph). The final temperature trace for the core c is then computed by combining the *dominating* $T^d(t)$ and *non-dominating* components, $T^{nd}(t)$ as follows:

$$\tau = \mathbf{h}(\mathbf{g}(b_1, \mathsf{M}, c), \cdots \mathbf{g}(b_n, \mathsf{M}, c)) \overset{\text{def}}{=} T^d + T^{nd} \tag{3.3}$$

with:

$$\begin{array}{rcc} \mathsf{B} & = & \{b_1, \cdots, b_n\} \\ T^d(t) & = & \max\limits_{b \in \mathsf{B}}(\mathbf{g}(b, \mathsf{M}, c)(t)) \\ T^{nd}(t) & = & \sum\limits_{b \in \mathsf{B}'} (\mathbf{s}(\mathbf{a}(b), \mathbf{v}(b)) \times \mathbf{g}(b, \mathsf{M}, c)(t)) \\ B' & = & B \setminus \left\{b \mid \mathbf{g}(b, \mathsf{M}, c)(t) == T^d(t)\right\} \end{array}$$

where $\mathbf{g}(b, \mathsf{M}, c)(t)$ is the value of temperature trace $\mathbf{g}(b, \mathsf{M}, c)$ at time instant t, $\times$ is the scalar multiplication operator, and $+$ is the point-wise addition operator. The proposed structure of (3.3) is justified by the following observations:

1. The temperature of an active core c is largely determined by the application executing on itself, dominating the observed temperature changes for that core;

2. If the core c is inactive, the temperature change on it cannot be lower than the maximum change in the temperature of c due to all other active cores acting individually;

3. The flow of heat between a pair of cores is proportional to the temperature difference between them; and

4. A certain power and temperature penalty is incurred when the first resource intensive application mapped to the processor invokes the on-board resource (e.g., power and temperature) management infrastructure. The penalty is amortized as more applications are mapped to the processor. This results in a large temperature change due to the first application, whereas further rise in temperature is relatively modest as more resource intensive applications are mapped to the processor, see Figure 3.2. This behavior is captured by the damping function **s**.

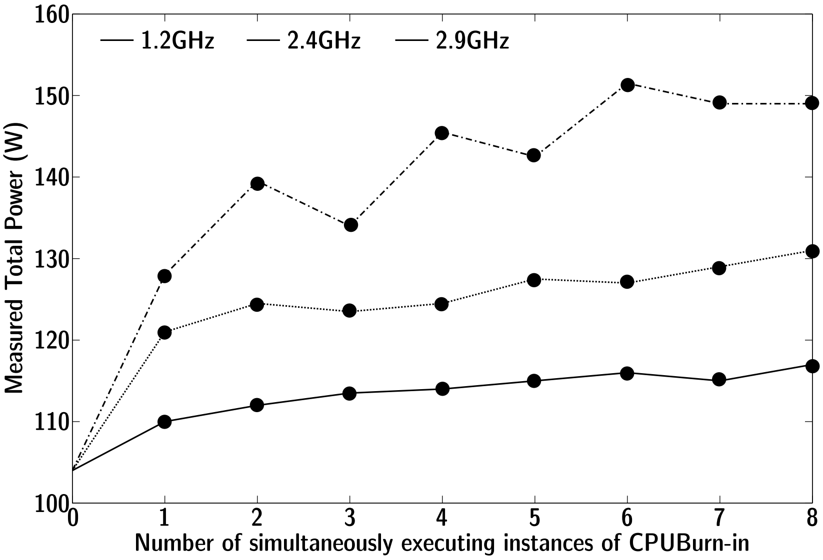


Figure 3.2: Measured power when increasing number of instances of the `CPUBurn-in` application are executed on the processor. Notice that the largest increase in the measured power consumption is due to the first instance.

Though (3.3) and its justification is based on the observations made on the Intel Xeon processor, the structure of (3.3) is expected to remain the same across multicore platforms. Specifically, observations 1, 2 and 3 listed above are due to thermal properties of silicon, independent of the processor make. Observation 4 is relevant for state-of-the-art processors which implement power and/or temperature management schemes. In case of a processor which does not implement such power and/or temperature management schemes, (3.3) can still be used (e.g., in an automated work-flow), as the damping function **s** will always return 1.

The Damping Function $\mathtt{s}$

The damping function $\mathtt{s}(a, v)$ is also determined using a sequence of calibration experiments as shown in Algorithm 6. The damping function is determined separately for each core type and application. The overall approach is to execute an application $a \in \mathsf{A}$ on a core type v located at the approximate center of the distribution of cores of the same type (line 2). A binding b is constructed in order to execute a at the highest possible operating clock speed f^* with the utilization trace u^* developed earlier (line 5). Selection of the highest clock speed improves the signal-to-noise (SNR) ratio permitting accurate estimates of the damping function. Next, the same application a is executed on *all* the cores of the similar type as per the same utilization trace u^* as before (line 10). The change in temperature on c^* is measured and compared against the expected change in temperature using the thermal model M yielding the damping ratio (line 15).

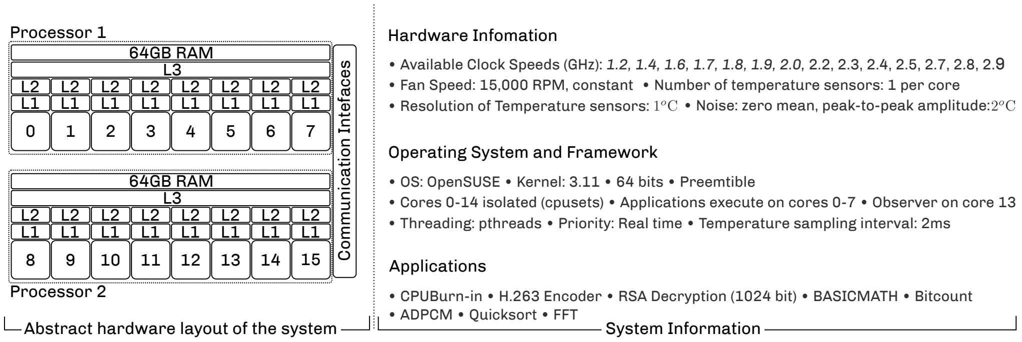
3.5.2 Limitations to the Accuracy of the Thermal Model

The accuracy of the thermal model and therefore the temperature traces may be limited due to the following factors:

1. The noise in the temperature sensors used during the calibration phase;
2. If the processor has a limited set of temperature sensors (e.g., less number of sensors than the number of cores) then there will be a few cores for which the true estimate of the temperature will not be available, also restricting the accuracy of the resulting thermal model;
3. If the designer chooses to trade-off the complexity of the model for speed, then the resulting model may not faithfully reproduce temperature traces, especially transients; and
4. Two temperature sensors located at an equal distance d from an active core c may not sense the same change in temperature as a temperature hotspot on c may not be symmetrically located. Algorithm 5 does not consider such asymmetry in order to keep the procedure and the model simple. However, should it be necessary, removing this inaccuracy from the model is straightforward.

```
   Input: Applications $\mathsf{A}$, Processor $\mathsf{P}$, Thermal Model $\mathsf{M}$
   Output: Damping Ratios $\mathbf{s}(a, v), \forall a \in \mathsf{A}, \forall v \in \mathsf{V}$
   Data: $f^* \leftarrow \max(\mathsf{F})$;                  // Choose highest operating clock speed
   Data: $t_s \leftarrow$ Discrete Sampling Interval;
   Data: $\tau_{raw}, \tau_{base}, \tau_{obs}$: Variables;
 1 $b \leftarrow \langle \phi, \phi, \phi, \phi \rangle$;                           // Initial binding, all empty
 2 foreach core type $v \in \mathsf{V}$ do
 3     Choose host core $c^* | \nu(c^*) = v$ s.t. $c^*$ is at the center of distribution of cores of similar type;
 4     foreach application $a \in \mathsf{A}$ do
 5         $b = \langle a, u^*, c^*, f^* \rangle$;                      // reuse u*from Algorithm 5
 6         Execute the binding $b$;
 7         $T_{raw} \leftarrow$ Observed temperature trace from core $c^*$;
 8         $T_{base} \leftarrow$ Denoise ($T_{raw}$);                                  // Baseline
 9         $\mathsf{C}^* \leftarrow \{c \mid c \in \mathsf{C}, \nu(c) = v\}$;
                       // Set of all cores of type $v \in \mathsf{V}$ including $c^*$;
10         $\mathsf{B}' \leftarrow \{\langle a, u^*, c, f^* \rangle \mid c \in \mathsf{C}^*\}$;
                                    // A set of bindings for all cores $c \in \mathsf{C}^*$
11         Execute the set of bindings $\mathsf{B}'$ ;
12         Execute the binding $b$;                   // See line 6;
13         $T_{raw} \leftarrow$ Observed temperature trace from core $c^*$ ;
14         $T_{obs} \leftarrow$ Denoise ($T_{raw}$) ;
15         $\mathbf{s}(a, v) = \frac{T_{obs} - T_{base}}{\sum(\mathbf{g}(b_1, \mathsf{M}, c^*), \cdots, \mathbf{g}(b_n, \mathsf{M}, c^*))}$;
                                                              // $\{b_1, \cdots, b_n\} = \mathsf{B}'$
16     end
17 end
19 Function Denoise($T_{raw}$)
                                                  // See Algorithm 5, line 49
20     return Denoised temperature trace;
```

Algorithm 6: Algorithm to compute the damping function.



Hardware Infomation

- Available Clock Speeds (GHz): *1.2, 1.4, 1.6, 1.7, 1.8, 1.9, 2.0,* 2.2, 2.3, 2.4, 2.5, 2.7, 2.8, 2.9
- Fan Speed: 15,000 RPM, constant • Number of temperature sensors: 1 per core
- Resolution of Temperature sensors: $1^o\mathrm{C}$ • Noise: zero mean, peak-to-peak amplitude: $2^o\mathrm{C}$

Operating System and Framework

- OS: OpenSUSE • Kernel: 3.11 • 64 bits • Preemtible
- Cores 0-14 isolated (cpusets) • Applications execute on cores 0-7 • Observer on core 13
- Threading: pthreads • Priority: Real time • Temperature sampling interval: 2ms

Applications

- CPUBurn-in • H.263 Encoder • RSA Decryption (1024 bit) • BASICMATH • Bitcount
- ADPCM • Quicksort • FFT

Figure 3.3: Hardware and software environment

Experimental results show that the actual loss of accuracy due to such factors is limited. Specifically, experimental results indicate that temperature traces estimated using the model M indicate an error which is of the same order as the quantization error in the on-chip temperature sensors. However, the error due to asymmetrically located hotspot with respect to temperature sensors may be observable (factor 4), specially when multiple applications execute simultaneously, see Figure 3.6.

3.6 Experiments and Results

Hardware Environment

Though the proposed approach for constructing the thermal model is versatile and largely independent of any specific hardware architecture, we particularly target a state-of-the-art processor featuring power and temperature management algorithms. Accordingly, a commercially available blade server consisting of two Intel Xeon E5-2690 8-core processors sharing 128GB RAM was selected. The selected processor is not specifically designed for embedded applications, however, it is representative for the trend in using multicore and multiprocessor architectures in high-performance embedded applications. In the available system, cores 0-7 belong to processor with ID 1, whereas cores 8-15 belong to the processor with ID 2, see Figure 3.3. Though processors 1 and 2 share the same housing, experiments show that the heat flow between the processors is limited, unless processor(s) are generating significant amounts of heat (e.g., all the cores of the hot processor(s) are executing some computationally intensive application). Usually, this is not the case, and given that the temperature sensors are noisy, the thermal effect of a processor on the other is difficult to detect. Therefore, in this chapter, we assume that the processors are thermally isolated from each other. The automatic on-demand clock speed increase (i.e., the *Intel turbo boost*) was disabled in order to prevent any unplanned operating clock speed changes. All cores within *each* processor can operate only at a *single* common clock speed in the range 1.2GHz - 2.9GHz. Precise control over the system fans is not possible, therefore, in order to avoid any errors due to unforeseen variation fan speeds, all fans operate at a constant full speed of 15,000 RPM. The processor features 1 temperature sensor per core with a resolution of 1^oC and error of $\pm1^o$C.

Software Environment

The operating system (OS) is a preemptible Linux kernel 3.11, contained to core 15 using `cpusets`, see [DJL+04]. All non-essential kernel services are stopped in order to prevent any unexpected variations in temperature or resource availability during the experiments. A dedicated observer on core 13 utilizes a modified `coretemp` module for reading temperature sensors every 1 ms, which represents the common tick interval in modern systems. Each application executes with real-time priority according to a precomputed binding. All algorithms are implemented in Matlab (R2013b) and the system identification toolbox is used to construct the thermal model (line 31, Algorithm 5). A mix of eight applications `CPUBurn-in`, `H.263 Encoder`, `Basicmath`, `RSA Decoder`, `Bitcount`, `ADPCM`, `Quicksort` and `FFT` were carefully selected in order to represent a wide variety of embedded applications, see [Mic12, LPMS97, Off13, GRE+01a].

The Results

The validity of the proposed approach is justified by the following results:

1. *Accuracy:* The error in the temperature traces estimated using the thermal model `M` is less than 2^oC (the overall error in the temperature sensors) for all tested bindings, including those in which a core switches between applications and clock speeds at runtime. This can be verified from the reported values of mean squared errors (MSE);
2. *Use case:* An example of temperature aware DSE in order to reduce the thermal cycles experienced by a given core; and
3. *Efficiency:* Memory and computational complexity of computing temperature traces at the DSE stage showing the resource efficiency of the developed thermal model.

Accuracy

Due to the combination of sensor noise and high thermal resistivity of silicon, we restricted the distance d_{max} from the hot core upto which the corresponding thermal effects are measured, i.e., $d_{max} = 3$, see Section 3.5.1

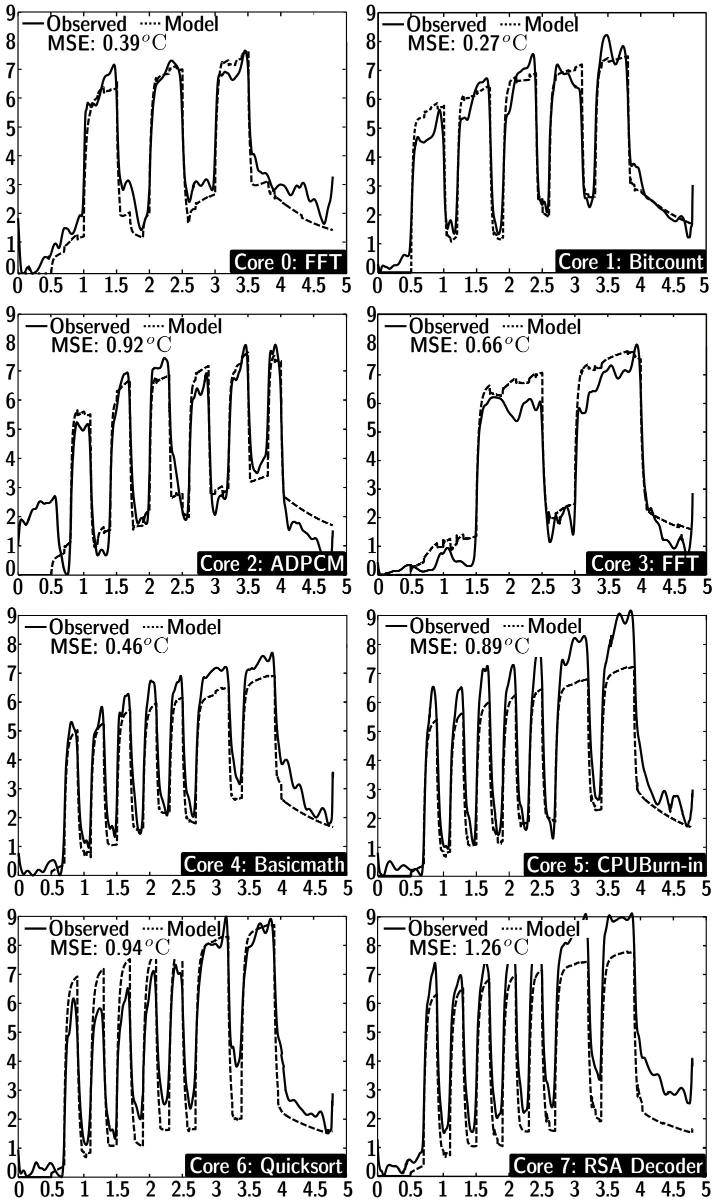


Figure 3.4: Accuracy of the thermal model when *each* core executes an application according to a predefined binding.See Figure 3.6 for more details. Clock speed: 2.9GHz. Horizontal axis: time in seconds; Vertical axis: Change in temperature, °C.

and Algorithm 5. This also reduced the number of experiments while retaining the required accuracy during the calibration and validation experiments. Three types of experiments were carried out:

1. *Multiple applications:* 64 experiments were designed, out of which the first 23 experiments required all the cores in the range 0-7 to execute a mutually unique application from the set A. For the next 41 experiments, a randomly chosen subset of cores execute randomly

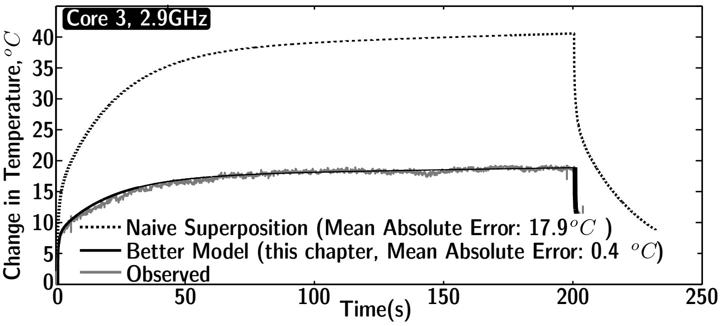


Figure 3.5: Long term accuracy of the thermal model when all cores execute a different application from the set A. Note that the metric used is Mean Absolute Error.

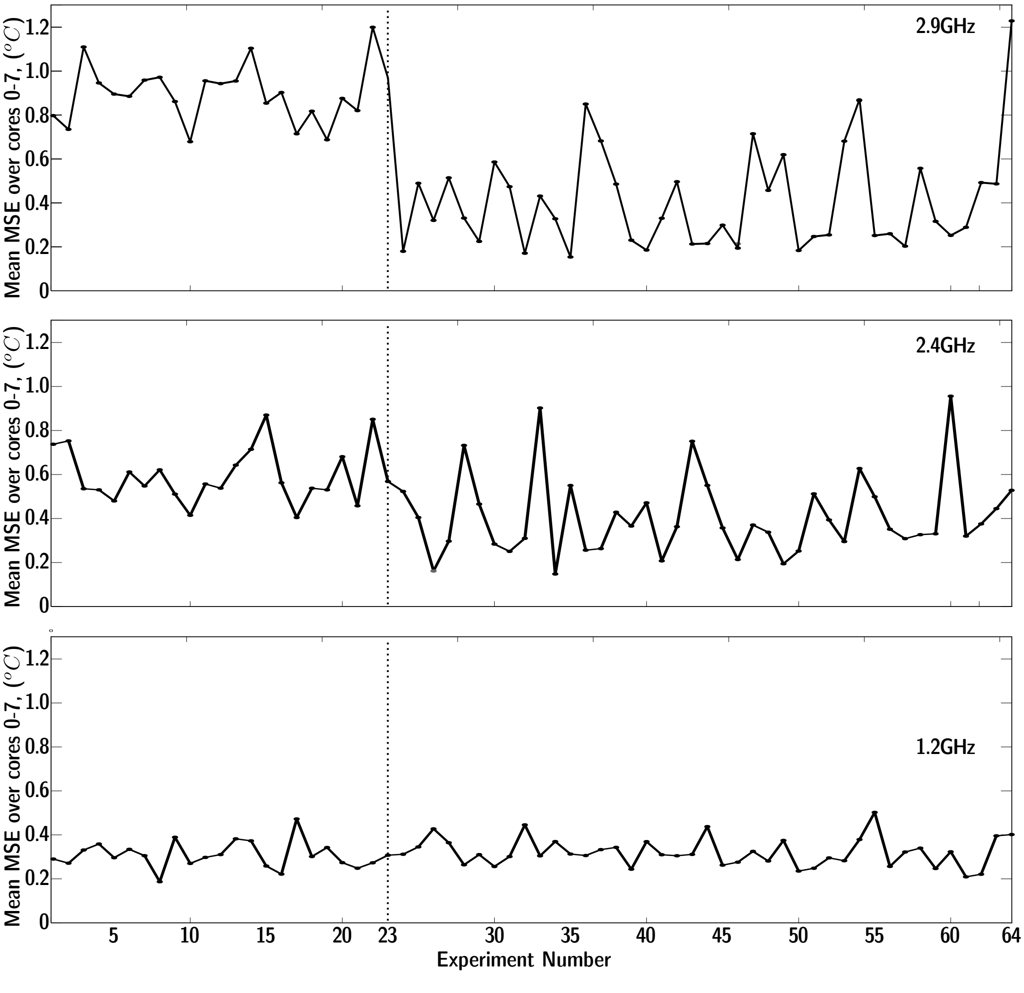


Figure 3.6: Summary of 64 experiments for each processor clock speed in 2.9GHz, 2.4GHz, and 1.2GHz. Each experiment from 1-23 requires all 8 cores to execute a mutually unique application. Each experiment from 24-64 selects a random set of cores and applications. Applications do not share cores.

selected applications. Applications do not share cores. The associated utilization traces were constructed such that the continuous execution time of the corresponding task was a random variable in the range 100-180,000 ms. The experiments were repeated for processor clock speeds in the set {2.9GHz, 2.4GHz, 1.2GHz}.

2. *Dynamic task and clock speed switching:* a core multitasks between applications and simultaneously cycles between operating clock speed in the set {2.9GHz, 2.4GHz, 1.2GHz}.
3. *Long Range:* Cores 0-7 executes a randomly drawn application from the set A for 200s.

Detailed temperature trace comparisons are provided for a subset of results, whereas a full summary is presented in Figure 3.6.

The mean squared error (MSE) in the temperature traces computed using the model remains less than 1.5°C, which approximates the uncertainty level in the temperature sensors, even when the temperature traces have significant transients, see Figure 3.4. Notice that the continuous execution time of tasks in Figure 3.4 is restricted in order to present sufficient visual details on the accuracy of the thermal model in estimating temperature transients. As a consequence, the dynamic range of the observed temperature traces is less than 10°C. Finally, the mean absolute error in the estimated temperature trace when each core (0-7) executes

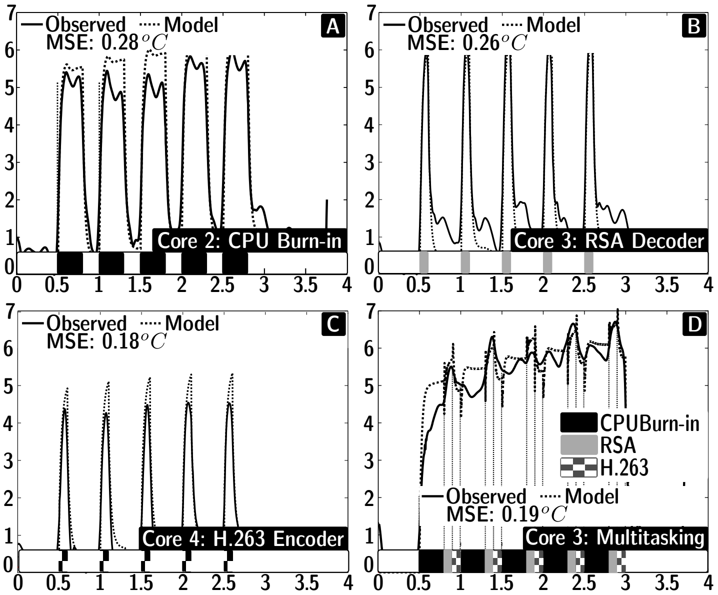


Figure 3.7: Thermally-aware DSE: Figure A-C show thermal cycles experienced by the processor for a naive binding. Figure D shows an optimized binding which reduces thermal cycles. The total work done by the three tasks remains the same. Horizontal axis: time(s); Vertical axis: Change in temperature, °C.

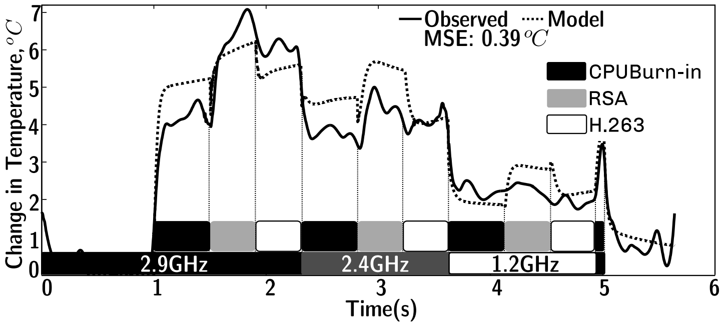


Figure 3.8: Estimated vs. observed temperature for a core with dynamic clock speed and application switching.

a randomly drawn application from the set A for hundreds of seconds remains small (< 1^oC), see Figure 3.5. For reference, the error due to a (naïve) lumped thermal model is also shown. The correlation between errors across different clock speeds is due to an error in estimated thermal model for the `BASICMATH` application due to the asymmetrically located temperature hotspot, as discussed in Section 3.5.2, see Figure 3.6. The temperature traces estimated using the model are accurate even when a core dynamically changes between clock speed and tasks, with the observed mean squared error being 0.4^oC, see Figure 3.8.

Use Case for Temperature Aware DSE

Reducing thermal cycles experienced by the processor may be one of the constraints which must be met by a feasible solution. Consider the case wherein three tasks execute on different cores with given schedules, but result in significant thermal cycles, see Figure 3.7 (A-C). This may be common when the tasks are periodic in nature, and their average utilization of the given core is less than 100%. A temperature-aware DSE can be used to explore better bindings which can significantly decrease thermal cycles experienced by the cores in the processor, see Figure 3.7 (D). Notice that the total amount of work done by each application in Figures 3.7 (A-C) is exactly the same as in Figure 3.7 (D). In the presented example, the new binding was obtained as follows:

- Pick an initial hot core c which features a low utilization, $u < 1$;
- Move an application to the core c from another hot core c' with the utilization u' if the following relationship is satisfied: $u + u' \leq 1$; and
- Repeat until no more applications can be moved to c.

If $u < 1$ after the procedure, then the thermal cycle on core c may be further reduced by lowering the clock speed of the core in order to bring core

c's utilization as close to 100% as possible. However, this step was not performed in the presented use-case, as the new binding had a utilization of 100% on core 3, see see Figure 3.7 (D).

Memory and Computational Costs for DSE

The memory and computational requirements for estimating the temperature traces directly depend on the complexity of the assumed thermal model. For the complexity of [nb = 5, nf = 5, nk = 2] used in this work, the corresponding *fit* obtained was 88%. Each new temperature data point requires $(\mathrm{nb}+\mathrm{nf})$ multiplications, 1 division and $(\mathrm{nb}+\mathrm{nf})$ additions, with the associated storage requirements for $(\mathrm{nb}+\mathrm{nf})$ coefficients and $(\mathrm{nk}+\mathrm{nb}-1)$ data points for the delay operator. Therefore, it can be concluded that the memory and compute costs incurred in using the thermal model during DSE with state-of-the-art processors is insignificant.

3.7 Closing Remarks

This chapter presented a calibration based technique to construct a fast and accurate thermal model for a state-of-the-art multicore processor, without requiring any auxiliary information such as the processor floorplan, or detailed power traces, making the proposed technique applicable to any processor. Extensive experiments on an Intel Xeon 8-core processor with 8 applications validate both the accuracy and the speed of the proposed technique. Specifically, it was shown that the error in the estimated temperature traces using the proposed technique is within the noise envelope of the on-chip temperature sensors used during the calibration phase, for all tested bindings, including those in which all cores of the processor execute a different application. A use-case was presented in which the thermal model M of the processor is used to obtain thermally favorable bindings allowing the processor to run without any danger of experiencing thermally induced faults or unforeseen performance losses.

Furthermore, all the calibration experiments can be performed without requiring any physical access to the processor, and without the need for any external instrumentation. Therefore, it is possible to generate the thermal model of systems (i.e., servers, notebooks, mobile, and even embedded devices) remotely, or on the field. Such a capability can be exploited by system vendors to generate accurate thermal models of their systems post-production, in order to account for the inevitable differences in thermal properties between the processors owing to design, manufacturing and other variations. Remote generation and recalibration of thermal models can also be used to continuously update the thermal models of the system when the field environment changes (e.g., upgrades to the system chassis), so that the thermal management algorithms can continue to operate effectively over the entire lifespan of the system.

4

Incorporating the Processor Cooling System into the Model

Summary

This chapter presents a method to construct fan-aware system thermal models for air-cooled state-of-the-art systems. Previous chapters have considered processors which are cooled at some constant rate. However, instead of constructing a completely new, monolithic fan-aware thermal model for the complete system, we take divide-and-conquer approach of proposing a thermal model of *only* the system fan, which can be cascaded to models obtained in the previous chapters. In line with the theme of previous chapters, the proposed method does not require any power trace information, nor the knowledge of physical properties of the system fan, and also does not require physical access to the system.

4.1 Introduction

Thus far, this thesis has presented a calibration based technique for constructing a thermal model for a given state-of-the-art processor which features one or more on-chip temperature sensors. Furthermore, previous chapters have only implicitly considered the presence of system fan(s) by assuming some constant rate at which the processor is cooled. Whereas one or more system fans in a server or a notebook computer is normal, recent mobile and high performance systems with ever more powerful processors have also started to feature a system fan in order to keep the system temperature within safe limits. One example is the Parallella multiprocessor, which was originally shipped without a fan, but is usually retrofitted with it, see [And13]. Another example is the Microsoft Surface Pro series of tablet computers featuring the Intel Core-i7 processor and a system fan, see [Jar14]. Even in traditionally fan-less systems, such as the automotive Electronic Control Units (ECUs), a fan may become mandatory to cool the processor as more compute intensive functions (e.g., vision, radar, collision avoidance, multimedia) are packed into a given system, see [PSHA03].

Given that these systems are constrained either by the available energy source (i.e., a battery) or the available computational capacity, system designers may use Design Space Exploration (DSE) tools to determine the best application mappings, scheduling, processor speed scaling, and cooling options (together, *binding*) for meeting the performance objectives while satisfying temperature and energy constraints. To this end, the selected DSE tool must be *fan-aware*. In other words, the thermal model of the system fan (succinctly, the *fan thermal model*) must be included with the temperature estimation algorithm(s) used.

Fan thermal models have commonly been used for designing power efficient cooling solutions for server class systems. Accordingly, most approaches are specifically designed for such systems, and require detailed system power traces which are obtainable via onboard power sensors, or external instrumentation. However, in contrast to server class systems, notebooks, mobile and high performance embedded systems do not usually feature power sensors, thus ruling out power trace based approaches for constructing fan thermal models. Although it is possible to collect power traces and other relevant information by instrumenting a disassembled device, the corresponding thermal properties may vary significantly from the original, resulting in an invalid thermal model. Mobile devices featuring a smart battery interface may provide access to system level power traces, but unless the portion of total power consumed by the CPU, the dominant heat generating component, can be accurately estimated, any thermal modeling approach relying on power trace information may not be sufficiently accurate, see [DZ11]. In principle, one may use the Hotspot thermal simulator which supports modeling the system fan, see [SSS$^+$04a]. However, it requires access to information which may not be readily available, e.g., floorplan of the processor, detailed power traces and knowledge of the physical properties of the system fan.

In order to overcome these challenges, a calibration based method to construct the fan thermal model is proposed, which differs from the state-of-the-art in three important aspects:

1. We acknowledge that developing a holistic fan-aware thermal model of the system may be infeasible as all the factors influencing the overall temperature of the system (e.g., applications, associated mappings and schedules, processor clock speeds, fan speed) must be accurately

accounted for. Instead, we follow the idea of separation of concerns, and take the simpler approach of constructing the fan thermal model which can be cascaded to the existing fan *un*aware thermal models available in the current DSE tools;

2. By utilizing onboard temperature sensors, we eliminate the need to acquire power traces, thermal properties of the heatsink, or the knowledge of the physical properties of the system fan; and
3. Lastly, since the proposed approach is purely software driven, it is feasible to (re)construct the fan thermal model *in-situ* for a device (e.g., a tablet computer such as the Microsoft Surface Pro) in the field, or even automatically when hardware upgrades (e.g., new fans) are carried out.

Accordingly, the fan aware thermal model is obtained as follows:

1. Use a fan-unaware thermal model which estimates temperature traces T_{nofan}; and
2. Cascade the fan *un*aware thermal model with thermal model of *only* the fan. The latter accepts T_{nofan} and fan speed s as inputs, and computes temperature traces which include the effect of the system fan. Here, a *temperature trace* refers to a time-ordered sequence of temperature measurements or estimates at some time resolution t_s.

In line with the previous chapters, the strength of the approach is the ability to estimate the fan thermal model without requiring detailed knowledge of system parameters. Additionally, a cascadable fan thermal model is relatively simpler to construct as compared to a monolithic fan aware system thermal model. We show that such separation of concerns is indeed feasible (see Figure 4.1 for the physical interpretation), and the individual solutions can be combined to yield an accurate fan aware thermal model of the system.

Statement of the Problem

The problem solved in this chapter can be summarized as:

Given:

1. A computer system with at least one temperature sensor and a user controllable fan;
2. Temperature traces T_{nofan} computed by a fan-unaware temperature estimation method.

Estimate: The thermal model of the fan $\mathbf{h}(s)$ such that the temperature trace T_{fan} incorporating the effect of the fan can be computed as follows:

$$T_{fan} = T_{nofan} \otimes \mathbf{h}(s) \tag{4.1}$$

where $\otimes$ is the standard convolution operator, and $\mathbf{h}(s)$ is the proposed thermal model of the fan as a function of fan speed s. We assume that the fan can be modeled as (an approximately) linear thermal system. The justification comes from observing the effect of the fan on the temperature traces during the calibration step (see Section 4.3.3), and is confirmed during experiments where it is shown that a linear fan thermal model estimates the temperature traces with sufficient precision (see Section 4.4).

The proposed thermal model of the fan is agnostic to the internal details of the fan-unaware thermal model, but in the specific case wherein the system does not provide information such as detailed power traces conventionally considered critical for a fan-unaware thermal model, we assume the availability of a power-agnostic, fan-unaware thermal model which can compute T_{nofan} without requiring the knowledge of system power traces or any physical parameters, relying solely on the given mapping information associated with each candidate solution, see Chapters 2 and 3.

4.2 Setup and Notation

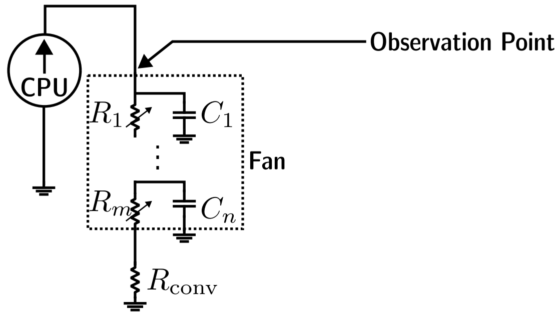


Figure 4.1: The equivalent RC-style model of the fan

In the interest of brevity, the following terms are carried over from previous chapters, and will not be redefined: (i) chip muliprocessor P, (ii) set of processor clock speeds F, (iii) processor core types V, (iv) set of applications A, (v) sampling time interval t_s, and (vi) utilization trace U. We also assume that a set O of temperature sensors are available on the chip. A temperature trace generated by the sensor $o \in \mathsf{O}$ is denoted as a tuple $T_o = \langle \tau_0, \tau_1, \cdots, \tau_i, \cdots \rangle_o$ where τ_i represents the temperature sample at the time instant $t = i$. Accordingly, a fan-unaware thermal model estimates the temperature trace $T_{nofan} = \{T_0, \cdots, T_{|\mathsf{S}|-1}\}$ for each candidate solution to the given a set of bindings. A single system fan which can operate at a user-determined speed s from a set of possible discrete speeds S, i.e., $s \in$ S is available to cool the system. The binding has a new variable, the fan speed s, with a binding $b = \langle a \in \mathsf{A}, u \in \mathsf{U}, c \in \mathsf{C}, f \in \mathsf{F}, s \in \mathsf{S}\rangle \in \mathsf{B}$ indicating that an application a executes according to a utilization trace u on a core c which operates at a frequency f, with the system fan operating at a constant speed s.

Keeping with the convention used in the literature, we suppose that the thermal model of the system fan $\mathbf{h}(s)$ can be represented as several cascaded RC sections, whose response can be described by an n order transfer function, see 4.1. The speed of the fan influences the rate of flow of heat from the processor, and is modeled by variable resistors, $R_1 \dots R_m$. The thermal capacitance of the processor is determined by its physical package and remains unchanged. Finally, given a binding b, the temperature traces T_{nofan} computed by the fan-unaware thermal model are assumed equivalent to the temperature traces *observed* when the processor P executes b with fan speed set to s_0. In principle, s_0 can take any value in S. In the current chapter, we take $s_0 = 0$ RPM.

4.3 Computing the Thermal Model of the Fan

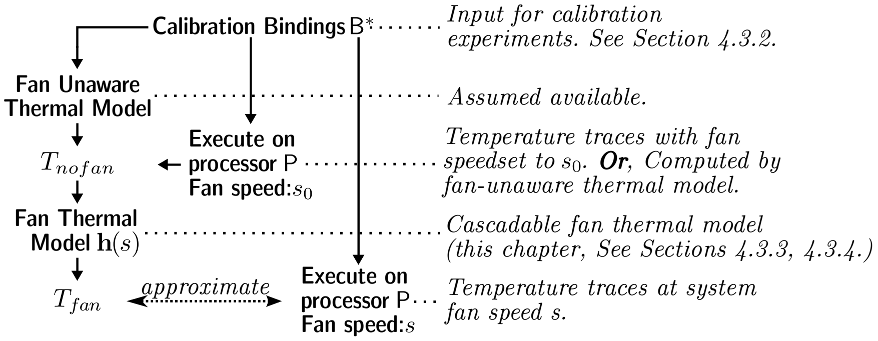


Figure 4.2: Overview of the approach.

4.3.1 Overview of the Approach

The fan thermal model $\mathbf{h}(s)$ computes the temperature traces accounting for the effect of the fan operating at a known speed s, given the temperature traces T_{nofan} computed by a fan-unaware thermal model. In particular, $\mathbf{h}(s)$ is a linear n-order discrete time transfer function of the form $\frac{N(z^{-1})}{D(z^{-1})}$ where $N(z^{-1})$ and $D(z^{-1})$ are polynomials in z^{-1}, the discrete time delay operator.

The procedure to compute the thermal model of the fan proceeds as follows:

1. Design a calibration trace $u^* \in \mathsf{U}$ and a set of special bindings $\mathsf{B}^* \subset \mathsf{B}$;
2. Obtain the associated temperature traces T_{nofan} which do not account for the system fan:
   (a) **Either** execute B^* on the processor P with the fan speed set to s_0, **or**
   (b) Obtain T_{nofan} from the fan-unaware thermal model by supplying it with B^*;
3. Obtain T_{fan} by executing B^* on the processor P with the fan speed set to $s \in \mathsf{S}$;
4. Find $\mathbf{h}(s) : T_{fan} \approx T_{nofan} \otimes \mathbf{h}(s)$, see (4.1). The approximate relationship relaxes the constraints on $\mathbf{h}(s)$ by requiring that the traces computed using $\mathbf{h}(s)$ approximate T_{fan}, instead of strict equality;
5. Reduce the order of the model $\mathbf{h}(s)$.

4.3.2 The Calibration Trace, u^*

The design of the calibration trace is similar to the design proposed in the previous chapter, with the exception of a new *Bias Segment*, see Section 3.5.1. Thus the calibration trace consists of three segments: the *Bias Segment*, the *Dynamics Segment*, and the *Statics Segment*. The *Bias Segment* is the first section of the calibration trace in which the application *does not execute*. The purpose of this section is to observe the ambient temperature *inside the system* which is significantly influenced by the fan speed. The notions of *Dynamics* and the *Statics* segments are similar to the proposal in Chapter 3, and are not being repeated again.

4.3.3 Computing the n-order Transfer Function

Based on the separation of concerns principle followed in this work, the thermal model of the fan must be *solely* a function of the fan speed, and therefore, any set of bindings can be used in order to obtain T_{nofan}. However, as the temperature sensors on-board any practical system are usually noisy, the accuracy of the model is improved by using a set of specifically designed set of bindings which maximize the signal-to-noise (SNR) ratio available as an input to the model estimation step. Hence an application $a^* \in \mathsf{A}$ is chosen which leads to the highest peak temperature on the given processor P, and is executed at the highest operating frequency $f^* = \max(\mathsf{F})$. The SNR improves even further if all cores dissipate heat. Therefore, for each fan speed s, the specifically designed set of bindings $\mathsf{B}^* = \{b | b = \langle a^*, u^*, c, f^*, s\rangle, \forall c \in \mathsf{C}\} \subset \mathsf{B}$ is executed in parallel on the processor. Here, an *instance* of the application a^* executes according to the utilization trace u^* at the highest possible frequency f^* on each core c of the processor with the fan speed set at a constant speed s.

The algorithm is similar to the one proposed in Chapter 3, see Algorithm 5. However, the `EstimateModel` function takes an additional parameter $bias$, which is used to compute an offset to be applied to the estimated temperature traces. The $bias$ is a function of system fan speed, see line 10. Further details are presented in Algorithm 7.

Referring to Algorithm 7, the estimation of the model (lines 13 - 29) begins with computing the initial bias in the observed traces as compared to the baseline, see line 5. This bias is the difference in the ambient temperature as recorded by on-board sensors when the fan operates at a speed s as compared to the baseline (i.e., when the fan operates at the speed s_0). Next, the baseline trace T_{base} is collected by executing the binding B^* on the processor P^* with system fan speed set at s_0. The traces are then denoised using edge-preserving algorithms (e.g., Daubechies wavelets, [HG08]), see line 40. Next, for each fan speed $s \in \mathsf{S} \backslash s_0$, the *same* set of bindings B^* is executed. The collected trace is denoised, and then the tuple $\langle T_{\text{obs}}, T_{\text{nofan}}, t_s, u^* \rangle$ is used to compute the fan thermal model for the fan speed $s \in \mathsf{S} \setminus s_0$. The description of the `EstimateModel` function is similar to the description provided in Chapter 3, Algorithm 5, and is therefore not being repeated here. However, do notice the additional parameter $bias$ which is applied to the estimated temperature traces. Note that the computed n-order models can be mapped to the conventional RC style structure shown in the Figure 4.1.

4.3.4 Model Order Reduction

Since the fan dissipates heat preventing the processor from heating up as much as it would in the absence of it, the fan is equivalent to a large thermal capacitor attached to the processor via a variable resistor, see Figure 4.1. This conclusion is supported by the analysis of the thermal models $h \in H$ obtained from the calibration experiments, see Algorithm 7. Specifically, it was found that for each thermal model h, a single pole dominates the overall response of the given model. Therefore, by trading off some accuracy, the n-order models computed in Algorithm 7 can be reduced to a single pole model by discarding non-dominating poles. The parameters of the simplified single-pole model (specifically, the scalar constant and the dominant pole position) are then tuned to require the response of the simplified model approximate that of the original higher order model. Several algorithms to achieve such a transformation are available, and we skip a detailed discussion in this chapter, see [SMD14, HG08]. The proposed model order reduction step does not significantly deteriorate the accuracy of the model, with the worst case mean-squared-error (MSE) in the computed T_{fan} being less than 1^oC in our validation experiments, see Section 4.4 for further details.

```
Input: Processor P, Binding B$^*$
Output: Thermal Model $H$
Data: $t_s \leftarrow$ Discrete Sampling Interval;
Data: $T_{\text{raw}} = T_{\text{obs}} \leftarrow 0$, bias = 0;                    // local variables
$T_{\text{nofan}} \leftarrow$ temperature traces when P executes B$^*$ at fan speed $s_0$;
                        // Or, computed by fan-unaware thermal model, given B*
$T_{\text{nofan}} \leftarrow$ Denoise ($T_{\text{nofan}}$);          // Remove noise from observations
;
foreach fan speed $s \in$ S $\setminus s_0$ do
    $bias \leftarrow$ GetBias ($s$)              // The steady state bias at fan speed $s$.
    $\mathbf{h}(s) \leftarrow 0$ ;
    Execute B$^*$ at fan speed $s$;
    $T_{\text{obs}} \leftarrow$ Observed Temperature traces;
    $T_{\text{obs}} \leftarrow$ Denoise ($T_{\text{obs}}$);
    $\mathbf{h}(s) \leftarrow$ EstimateModel ($T_{\text{obs}}$, $T_{\text{nofan}}$, $t_s$, bias) ;
end

Function EstimateModel($T_{\text{obs}}$, $T_{\text{nofan}}$, $t_s$, bias)
                 // $T_{\text{obs}}$: Observed temperature trace, $t_s$: Sampling interval
     // $T_{\text{nofan}}$: temperature trace with fan speed $s_0$, $bias$: bias temperature.
    Data: [nb, nf, nk] $\leftarrow$ [2, 2, 1] ;
     // Initial order of model as vector [#zeros+1, #poles, #delay(samples)]
    Data: fit $\leftarrow -1$, fit$'$ $\leftarrow -1$;
                           // fitness of the estimated model, Maximum: 100%.
    Data: System Constraint: MAXPOLES, MAXZEROS, MAXDELAY;
    Data: $h$: Computed temperature model of type $\frac{N(z^{-1})}{D(z^{-1})}$ ;
    for nk = 1:MAXDELAY do
        for z = 1:MAXZEROS do
            for p = 1:MAXPOLES do
                Compute model $h' = \frac{N(z^{-1})}{D(z^{-1})} \mid T_{\text{nofan}} \otimes h' + e + \text{bias} \approx T_{\text{obs}}$
                               // $t_s$: sampling interval, also see [HG08]
                 // $e$ is the assumed noise model, e.g., zero mean white noise

                if $fit > fit'$ then
                    fit $\leftarrow$ ComputeFit($T_{obs}$, $T_{nofan} \otimes h' + e +$ bias);
                    fit$'$ $\leftarrow fit$ ;
                    $h \leftarrow h'$ ;
                                                   // the best model so far.
                    if $fit = 100$ then
                        return [$h$, bias];                     // perfect model.
                    end
                end
            end
        end
    end
    return [$h$ , bias];

Function ComputeFit($\hat{T}$, $\hat{T}^*$)
    return $100 \times \left\{ 1 - \frac{\|\hat{T}^* - \hat{T}\|}{\|\hat{T} - \overline{\hat{T}}\|} \right\}$ ;
      // Normalized Root Mean Square Error estimate. $\overline{\hat{T}}$: Mean value of $\hat{T}$

Function GetBias($s$)
    $T_1 \leftarrow$ mean steady-state temperature with no applications and fan speed set to $s_0$;
    $T_2 \leftarrow$ mean steady-state temperature with no applications and fan speed set to $s$;
    bias $\leftarrow T_2 - T_1$ ;
    return bias ;
                    // Returns the constant bias in the initial conditions.

Function Denoise($T_{raw}$)
     // Remove noise from the input trace using edge-preserving algorithms,
    e.g., Daubechies wavelets.
    return Denoised temperature trace;
```

Algorithm 7: Calibration based algorithm to compute the thermal model $\mathbf{h}(s)$. The structure of the algorithm is similar to Algorithm 5 except for an additional *bias* parameter.

4.3.5 Sources of Errors

Factors such as noise in the temperature sensors, limited number of sensors, and capping the order of the models all affect the overall accuracy of the estimated fan thermal model. These factors have already been discussed in Chapter 3, and are therefore not being reproduced again. Experimental results show that the loss of accuracy due to such factors is limited, with the worst case mean squared error to be about 1^oC. It is important to remember that the temperature sensors which are used to calibrate and validate the fan thermal model themselves have a quantization error of $\pm 1^oC$, limiting the maximum achievable accuracy.

4.4 Experiments and Results

The accuracy of the fan thermal model is validated by comparisons against corresponding traces observed on real hardware. A comparison against power-trace based approaches is avoided as these models are themselves approximate, and require inputs which may not be available (e.g., power traces for mobile devices). Instead, given that the proposed approach is based on sampling temperature sensors onboard the given testbed, the maximum achievable accuracy defined by the resolution and noise of the temperature sensors used.

4.4.1 Setup

Hardware

All experiments were carried out on a commercially available notebook computer with an Intel core-i7 processor consisting of 4 physical cores numbered 0..3, and 16GB RAM. The physical cores are linearly laid out, and hyper-threading allows each physical core to present two logical cores to the software environment. Each core features one temperature sensor with a resolution of 1^oC with an uncertainty (i.e., error) of $\pm 1^oC$. Therefore, we consider our thermal model to be accurate if the computed traces are within $\pm 1^oC$ of the reference observations. The processor can operate at a range of frequencies in [0.8-2.4]GHz, and at any instant, all cores must operate at one common frequency. A single system fan cools the processor, whose speed is controlled by setting appropriate values to an 8-bit PWM configuration register. The ambient temperature outside the notebook is maintained at 22^oC.

Software

The operating system is based on a preemptible Linux kernel 3.11 operating at runlevel 1 with only the most essential services. The temperature sensor driver (`coretemp`) is modified to read temperature sensors every 1ms, the most common tick-rate in the state-of-the-art systems. `Cpusets` are used to isolate physical cores 1-3 from any timing disturbances resulting from housekeeping and kernel tasks executing on core 0, see [DJL$^+$04]. The experimental framework is written in C which provides dedicated temperature and fan speed logging threads. A dedicated master-slave thread pair executes each given application as per a specified binding b. Physical cores 2 and 3 are reserved for executing applications both for calibration and validation, whereas logger threads execute on physical core 1. All algorithms are implemented in Matlab 2013b, with the system identification toolbox used for estimating models, see Algorithm 7.

For the experiments, the input T_{nofan} to the fan thermal model was obtained by executing a set of test bindings $\mathsf{B}' \subset \mathsf{B}$ with the fan speed set at $s_0 = 0$ RPM. The decision to obtain input traces both for calibrating and validating the fan thermal model from observations helps avoid accruing errors which may be present in the temperature traces computed by a fan-unaware thermal model. The reference traces for validation was obtained by re-executing B' at fan speeds $s \in \mathsf{S} \setminus s_0$, see Figure 4.2. Consistency between the traces obtained under two different fan speeds is ensured by

1. Limiting the number of simultaneously executing benchmarks on the physical cores 2 and 3 to 3 (= 6 threads). This reduces the timing jitters between master-slave thread pairs when a given experiment is repeated with two different fan speeds. Specifically, the approach reduces the latency between time instants at which a master thread dispatches a signal (`pause` or `go`) and the instants when the corresponding slave thread acts on it; and
2. Selecting (almost) deterministic applications.

These restrictions do not reflect any limitation of the conceptual approach, but are due to the experimental test and observation setup followed in this chapter.

The Application Set A

1. *CPU-Burnin*: Designed to stress power and thermal management solutions found in the x86 systems, see [Mic12];
2. *H.263 Encoder*: The encoder accepts input in a Quarter Common Intermediate (QCIF) format and produces an output stream at $\sim$ 30 fps. The encoder loops through 8 input files ranging from 8MB to 80MB in size, see [Mar14, LPMS97];

3. *RSA Decoder*: Built using the PolarSSL library, the decoder decrypts a text-stream originally encrypted with a 1024 bit key, see [Off13];

4. *ADPCM*: The modified application encoder-decoder chain processes one packet every $\sim$ 4ms. The encoded packet is 4:1 compressed and is 4KB in size, see [LPMS97];

5. *Blackscholes*: From the Parsec benchmark suite, see [Bie11].

4.4.2 Results

In order to validate the proposed approach, the following results are presented:

1. Observations of bias values at different fan speed settings;

2. Accuracy of the simplified single-pole model for short and long time test windows, with the associated mean-squared-errors (MSE). Results for short-time window experiments demonstrate the accuracy of the thermal model in estimating the transients, whereas the steady-state accuracy of the model is demonstrated by long-time window experiments;

3. Statistical summary of over 30 random experiments each for long and short time test windows, with the associated mean-squared-errors(MSE); and

4. Memory and runtime overhead.

Since the system fan cools the entire processor uniformly, and in order to present sufficiently detailed results, we specifically report the temperature traces for physical core 3 which runs the calibration and validation experiments (along with physical core 2). All conclusions apply equally to the temperature of all cores.

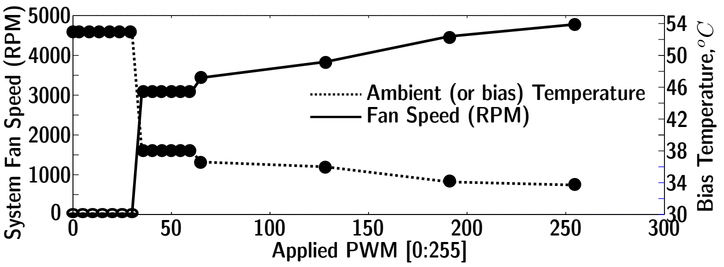


Figure 4.3: System Fan Speed vs Bias Temperature

Fan Speed and Bias Temperature

As expected, the speed of the fan significantly influences the ambient *inside* the system, see Figure 4.3. With the fan switched off, the ambient inside the system may reach as high as 54^{o}C, as compared to a relatively mild 34^{o}C when the fan operates at full speed. Also notice that the fan speed is not a smooth function of the applied PWM. The relationship between the applied PWM and the system fan speed is governed by hardware electronics and physical properties of the fan motor. As expected, the bias temperature is independent of the core operating frequency since cores do not compute when bias temperature observations are recorded, see Algorithm 7. Once the fan turns on, any further reduction in the ambient due to an increase in the fan speed is relatively modest.

Accuracy of the Simplified Single Pole Model

In order to demonstrate that the thermal model of the fan is indeed independent of any application, the model estimation algorithm uses *only* the `CPU-Burnin` application for calibration, see Algorithm 7. The validation experiments use one or more applications from A with bindings carefully constructed to test the accuracy of the model both at long time scales (several hundreds of seconds) and short time scales (several seconds).

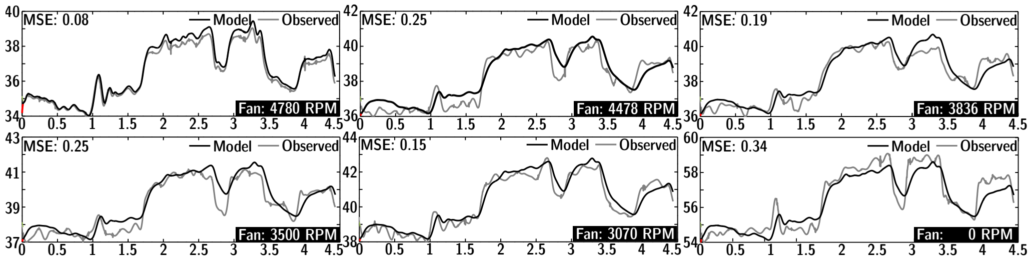


Figure 4.4: Accuracy of the fan thermal model with `H.263 Encoder`, `Blackscholes` and `ADPCM` mapped to cores 2 and 3. Horizontal axis: time(s), Vertical axis: Observed temperature (core 3), ^{o}C. All cores operate at 2.4GHz. Same experiment repeated for 6 fan speeds. Dynamic range is limited due to small window size. Note initial temperature.

Though the fan significantly influences the ambient temperature inside the system, the dynamic range of temperature experienced by the processor executing a given mapping remains relatively unaffected, at least when the task execution times are not very long, see Figure 4.4.

The impact of the fan on the dynamic temperature range experienced by the processor is clearer when the tasks have relatively longer run times (e.g., order of hundred of seconds), see Figure 4.5. For the shown example, the temperature change experienced by the processor is about 15^{o}C when the fan is operating at full speed, whereas when the fan is switched off, the change in temperature is about 25^{o}C. Also note that the temperature

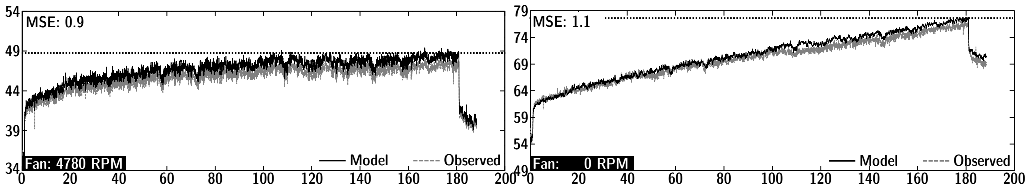


Figure 4.5: Accuracy of the fan thermal model vs the observed for long execution times of tasks mapped to the processor. Horizontal axis: time(s), Vertical axis: Observed temperature (core 3), ^{o}C. All cores operate at 2.4GHz. Notice the initial temperature which is significantly affected by the fan speed.

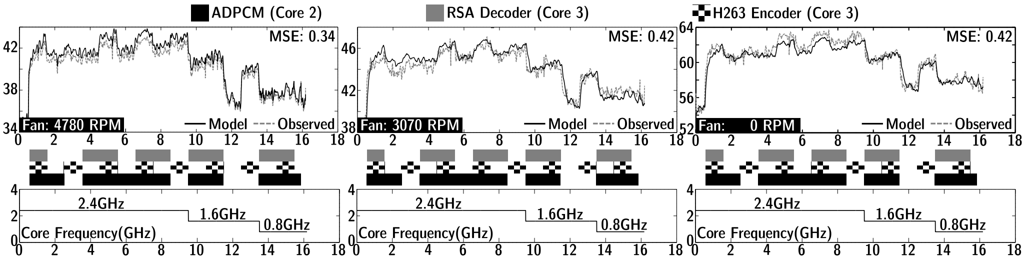


Figure 4.6: Accuracy of the fan thermal model when multiple tasks are mapped onto the processor under dynamic frequency switching. Horizontal axis: time(s), vertical axis: (top row): temperature,^{o}C; (bottom row): core frequency (GHz.)

in the former case can be seen as approaching an equilibrium, whereas in the latter case, the temperature will rise further before reaching a steady-state. The estimated thermal model of the fan is accurate to within the error in the temperature sensors, even when the processor dynamically changes its operating frequency, or when multiple tasks are mapped, thus validating the approach of the chapter, see Figure 4.6. Note that it was possible to execute the `RSA Decoder` and the `H.263 Encoder` on core 3 as hyper-threading was enabled.

Statistical Summary

Each point in each statistical summary table represents 3 experiments, see Tables 4.1 and 4.2. For each experiment, three applications were randomly drawn from the set A and pinned to physical cores 2 and 3. For long time window experiments, the schedule of each application is designed to ensure at least one continuous execution window of 200s. For short time window experiments, the continuous execution time of each application is a random variable between 20ms and 20s. All experiments were repeated for three processor clock frequencies and 6 discrete fan speeds (*c.f.*, Figure 4.3). The accuracy of both order-2 and order-1 fan thermal models are presented for each experiment. Gains in the accuracy with higher order models were not significant, and are therefore not presented. In general, and

Short Time Window Error Statistics: Average MSE for Order 2 model $(^o\mathrm{C})$						
Fan Speed (RPM) → Clock Frequency (GHz)↓	0	3070	3500	3836	4478	4780
0.8	0.64	0.44	0.12	0.2	0.18	0.52
1.6	0.56	0.56	0.2	0.51	0.61	0.72
2.4	0.67	0.46	0.56	0.52	0.63	0.73

Short Time Window Error Statistics: Average MSE for Order 1 model $(^o\mathrm{C})$						
Fan Speed (RPM) → Clock Frequency (GHz)↓	0	3070	3500	3836	4478	4780
0.8	0.32	0.3	0.37	0.27	0.24	0.33
1.6	0.35	0.6	0.36	0.49	0.58	0.64
2.4	0.38	0.92	0.73	0.91	0.64	0.97

Table 4.1: Statistical Summary: Short Time Windows

Long Time Window Error Statistics: Average MSE for Order 2 model $(^o\mathrm{C})$						
Fan Speed (RPM) → Clock Frequency (GHz)↓	0	3070	3500	3836	4478	4780
0.8	0.32	0.3	0.25	0.2	0.19	0.4
1.6	0.35	0.56	0.21	0.5	0.61	0.72
2.4	0.52	0.58	0.49	0.69	0.63	0.85

Long Time Window Error Statistics: Average MSE for Order 1 model $(^o\mathrm{C})$						
Fan Speed (RPM) → Clock Frequency (GHz)↓	0	3070	3500	3836	4478	4780
0.8	0.44	0.12	0.37	0.27	0.24	0.44
1.6	0.56	0.6	0.11	0.49	0.58	0.68
2.4	0.76	0.68	0.84	0.71	0.81	0.73

Table 4.2: Statistical Summary: Long Time Windows

as expected, a higher order model is relatively more accurate. The difference in accuracy is more pronounced at higher clock frequencies (2.4GHz) as there sharper transients where a higher order model performs better. The SNR deteriorates with lower clock frequencies which also nullifies the relative accuracy of a higher order model. However, for both long and short time window experiments, the average MSE remains below 1^{o}C, confirming the accuracy of both order-2 and order-1 models.

Compute and Memory Costs during DSE Exercise

Each single-pole model requires at most four values to be stored in memory: the positions of a pole and a zero, a scalar constant, and a bias value. Therefore, for a total of |S| distinct fan-speeds, the total memory overhead is $4 \times$ |S| variables. During DSE, computation of each temperature data point requires at most 3 multiplies and 3 additions, and storage space for two variables. It is important to note that even with such meager compute and memory overhead, it is possible to compute temperature traces due to the fan with a reasonable accuracy (i.e., MSE$<$ 1^{o}C).

4.5 Closing Remarks

This chapter presented a new calibration-based approach to constructing a thermal model of the given system fan without requiring access to power traces, or any physical information about the system of interest. In order to limit the overall complexity of the full fan-aware system thermal model, the proposed thermal model of the fan is designed to be cascaded to the already existing fan-unaware thermal model(s) in the current DSE tools. As expected, the thermal model of the fan is dependent solely on the fan speed and computes accurate temperature traces for all possible bindings of applications, core frequencies, schedules and fan speeds. The temperature traces computed by the model are accurate, with mean-squared-errors always less than 1^{o}C, noting that the quantization error in the temperature sensors is $\pm 1^o C$. Furthermore, we showed that the memory and compute costs incurred during the use of the fan model during DSE is indeed insignificant. Since the approach is purely software driven, the fan thermal model for a device can be (re)constructed on the field, such as when hardware upgrades are performed.

5

Estimating the Peak Temperature

Summary

This chapter presents an analytical approach to the estimate worst case peak temperature that may be experienced by a given processor. The approach requires an abstract description of the workload (e.g., applications, corresponding schedules), and abstract thermal properties of the processor which is to be used for executing the applications. The knowledge of the worst case peak temperature may be useful during design space exploration for eliminating those solutions (e.g., specific application and their corresponding scheduling combinations) which may lead to an overheated processor. It is common knowledge that even exhaustive simulations may not be sufficient to extract such worst case peak temperatures, except for trivially simple systems (e.g., the processor executes a single application continuously).

5.1 Introduction

Having obtained a thermal model of the processor, a system designer may wish to estimate the peak (or the worst case) temperature that may be experienced by the processor when it executes a given set of applications. For some of the simplest use-case scenarios, such as when the given processor executes a single application continuously, estimating the worst case temperature is trivial, and can be estimated directly by computing the step response of the models estimated in the previous chapters. However, in most practical use cases, an application's execution pattern is more complex, and may also include some timing uncertainties. Such execution patterns are abstractly described using the *period (P)*, *jitter (J)*, and *minimum inter-arrival distance (D)* model, with the interpretation as follows. An application is expected to execute periodically with a period P, i.e., it is expected to execute at time instants $t, t+P, t+2P, \cdots$. However, the *actual* time instants at which the application executes may vary upto within a time window J units wide, centered at the expected execution time. Furthermore, the same application may occasionally execute in a burst, in which case,the minimum time separation between two consecutive executions of the application is bounded by D. It must be pointed out that a given $\langle P, J, D\rangle$ tuple leads to a *set* of feasible execution patterns of the given application.

Clearly, estimating the worst case peak temperature for complex execution patterns as mentioned above remains challenging as all possible execution patterns may need to be explored. Of course, one option is to run exhaustive simulations in order to discover worst case peak temperature by simulating all possible execution patterns for each given binding (i.e., mapping between applications and cores, their corresponding schedules, processor clock speed, and if available, fan speed). It is clear that for most practical use case scenarios, the required number of simulations for may be intractable. Consequently, the system designer may choose to limit the number of simulations, risking the possibility that corner cases resulting in worst case peak temperatures are missed. Therefore, in this chapter, we turn to an analytical approach which can be used to quickly estimate the sought after worst case temperature of the processor, given an abstract description (e.g., task or application[1] execution times, task invocation periods, jitter in task arrivals, and resource availability) of the workload that is executed by it. The analytical method can be used to quickly eliminate those bindings which contain execution patterns that may overheat the processor. One may then proceed to use the remaining binding options together with the thermal model for an in-depth thermal investigation.

Therefore, this chapter attempts to answer the following question: *What is the worst-case peak temperature of a real-time embedded system under all feasible scenarios of task arrivals?* A new thermal-aware analytic framework is proposed that combines a general event/resource model based on network and real-time calculus with system thermal equations. This analysis framework has the capability to handle a broad range of uncertainties in terms of task execution times, task invocation periods, jitter in task arrivals, and resource availability.

We assume a simple, single pole thermal model for analysis: one which assumes that the entire processor is a point source of heat, i.e., a lumped

---

[1] This chapter uses the terms application" and "task" interchangeably.

thermal model. One may derive a single pole model by reducing the order of models obtained in the previous chapters, albeit incurring some errors. As an example, consider the thermal model derived for Xeon processor running the ADPCM application, with the processor clocked at 2.9GHz, with 3 poles and 3 zeros. It was possible to convert the given model to a continuous time model using zero-order hold approximation, and then to reduce the order of the model to contain only a single pole, making it suited to techniques discussed in this chapter, see Figure 5.1.

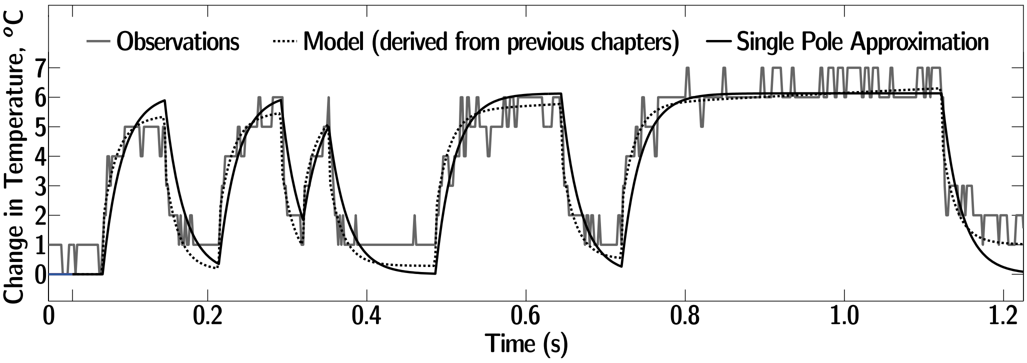


Figure 5.1: Approximation of a higher order discrete time model from previous chapters to a single pole continuous time model. Application: ADPCM, Processor: Xeon, Clock Speed: 2.9GHz. The original model describes a single application, i.e., ADPCM, executing on the processor.

The advantage of using the lumped thermal model is the possibility of obtaining a closed form solution to the thermal equation which is amenable to further analysis, see (5.9). Therefore, the simple model presents an opportunity to understand the fundamentals of interactions between discrete processes (e.g., discrete event arrivals) and continuous processes, such as temperature. The model considered in this chapter takes both dynamic and leakage power as well as thermal dependent conductivity into consideration. Though the assumed model is not in the form used in the previous chapters, as long as the adopted model is representative of the physics of heat transfer, theory developed in this chapter remains applicable when other thermal models are used, such as those obtained in the previous chapters.

Estimating worst case peak temperature in a manner that does not lead to significant overestimation is not trivial, except for very simple systems. For example, traditional real-time schedulability analysis is based on the critical instant of task releases to offer timing guarantees. However, for thermal investigations, even for simple arrival patterns such as periodic with jitter there is a lack of results about the critical instant which leads to the maximum peak temperature.

5.2 Simple Example

We illustrate the complexity of the above fundamental question by means of a simple example. Let us assume that we have a work-conserving component which processes just a single event stream which is periodic with jitter: period 120 ms, jitter 240 ms, computing time 30 ms, minimal inter-arrival time 30 ms. The thermal model of the processing component is equivalent to that used in Section 5.6. In order to explain the problem we are going to solve, let us compare five simple methods for computing the temperature:

1. We use the average workload of the tasks, i.e., utilization $U = 0.25$, and compute the corresponding average power of utilization 0.25 according to (5.6). The resulting steady state temperature T is 335.08 K.

2. We use a set of traces with random jitter that comply to the specification. For 100 independent runs with 1.2s trace length we find an average, standard deviation and maximum temperature of 350.98 K, 1.75 K and 354.46 K, respectively.

3. We use the well known critical instant, i.e., starting from an arbitrary but fixed time all task instances are released as early as possible. The peak temperature observed in this method is 351.63 K with a trace length of 1.2s.

4. We try to construct a reasonable critical instance as follows: We suppose that for an unlimited time, the task is periodic. At some time, the task shows its maximal jitter, i.e., a maximally dense workload pattern. In this case, the simulation shows a maximal temperature of 355.62 K.

5. We use the method developed in this chapter and get a tight bound on the maximal temperature of 359.22 K with a trace length of 1.2s.

It appears that none of the obvious methods is able to determine the maximal temperature even for such a very simple workload. The following Figure 5.2 illustrates the workload traces corresponding to options 4) and 5) as well as the corresponding temperature traces with an initial temperature of 319.31 K.

This chapter proposes an accurate system-level analytic technique which offers temperature-guarantees for real-time systems. When no information of the workload is given, the worst-case temperature can only be predicted assuming a fully stressed case, which results in the worst-case estimate of 402.33K. This is clearly too pessimistic and far from *tight* estimation. Information on the workload enables a tight peak temperature estimation as will be shown in what follows. We consider general event arrivals modeled by arrival curves in Real-Time Calculus [TCN00] and Network Calculus [LBT01a]. An arrival curve provides an upper bound (and a

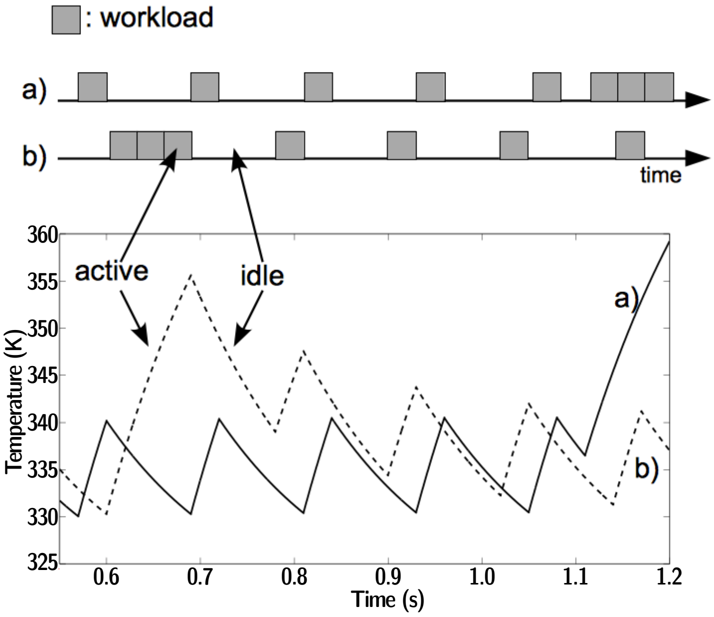


Figure 5.2: Simple example that shows (a) the worst case workload trace and (b) a constructed workload trace as well as the corresponding temperature changes.

lower bound) on the workload that might arrive to the system in any interval lengths. Usually, these curves can be derived by profiling sufficiently representative traces or by analyzing the specification of an application. Even though arrival curves constrain the possible workload injected to the system, there are infinitely many traces that comply to the provided bound on the workload, in terms of initial phase, jitter, or burstiness.

Another uncertainty we have in the system is the availability of the computing resource. A processor is not always completely available for computation in modern embedded systems due to dynamic resource management like dynamic frequency modulation. The same is true if active-idle schemes are used in order to reduce its peak temperature, i.e., switching the processor on and off using a predefined pattern. Thus, the computing resource provided by a processor within a given time interval can also be constrained by an upper and lower bound, denoted as service curve. An exhaustive search to check all possible combinations of workload traces and computation availabilities to determine the peak temperature of the system is infeasible.

As in most studies related to temperature analysis and simulation, cooling and heating of the system is modeled by means of Fourier's law, i.e., the law of heat conduction modeled by a linear differential equation.

The thermal behavior of processing architectures is usually modeled by considering heat conductances and capacitances of micro-architectural elements. It is also known that leakage power and thermal conductances are temperature dependent. In contrast to Fourier's law on cooling and heating, the resulting differential equation is no longer linear. Based on this generalized and more accurate thermal model, we use the technique proposed in this chapter to analyze the maximum peak temperature for given characterizations of the workload and the available computing resource under *workload-conserving* real-time scheduling algorithms, such as earliest-deadline-first (EDF), rate-monotonic (RM), deadline-monotonic (DM). The schedulability of a system can also be analyzed in the proposed technique in combination with existing Real-Time Calculus based schedulability analysis techniques [TCN00].

The contributions of this chapter can be summarized as follows:

- Based on a characterization of the task arrival variability as well as the availability of the computation resource, an upper bound on the peak temperature for any workload-conserving scheduling discipline is determined.
- The peak temperature analysis is applied to three commonly used power and temperature models that consider temperature dependent leakage and conductances into account.
- Extensive experimental studies validate the proposed analysis framework and show its applicability.

5.3 Related Work

Literature is available which explores the possibilities to reduce the peak temperature to meet performance constraints [BP05, CHK07, FCWT09], or maximizing performance under given peak temperature constraints, [BM01]. Approaches are also available which explore thermal control by applying control theory for system adaption [FWP09, FKC$^+$10, WMW09]. In general, none of the approaches which combine temperature and performance aspects explicitly estimate the worst case peak temperature that may be experienced by the processor.

First order linear differential equations have been traditionally used for thermal modeling of integrated circuits see Section 5.4.3 and [MMA$^+$07, WB06a, WB06b, WB08, ZC07]. This traditional model is simple, though inaccurate, as some parameters are vary with temperature which affects the accuracy of results significantly in deep sub-micron technology, see [LDSY07]. There are two temperature dependent parameters in the model: leakage power and thermal conductance, and it has been shown that both parameters have a quadratic dependency on the temperature of the processor, see [LHL05, LDSY07]. However, within the operating temperature ranges of current circuits, the leakage power can be accurately estimated by a linear model approximation also used in this chapter, see [LDSY07].

On the other hand, the variable thermal conductance makes thermal differential equation more complicated. This chapter includes temperature dependent leakage and conductance in system level temperature analysis. Using the principles outlined in this chapter, an extension of the presented worst case peak temperature analysis to multicore systems has been proposed, see [SBYT12]. Furthermore, traffic shaping techniques utilizing the proposed thermal model for designing temperature constrained systems have also been proposed, see [KT11].

5.4 System Model

We start with a simple lumped thermal model wherein leakage and thermal conductance parameters are held constant, and then include an advanced model which takes into account temperature dependent leakage and conductance parameters. Both models consider variability in the task arrivals, workloads, and computing resources.

5.4.1 Computational Model

The computational model of a processing component follows the ideas of network and real-time calculus. We suppose that the component receives in time interval $[s, t)$ a cumulative workload of $W(s, t)$ time units, e.g., in $[s, t)$ tasks with a total workload of $W(s, t)$ arrive. $W(s, t)$ is a stair case function for any fixed s, i.e., it has slope 0 almost everywhere and when a task arrives it jumps by its computation time. Similarly, the component is characterized by the availability of its computing resource, i.e., $R(s, t)$ time units are available for task processing in time interval $[s, t)$.

Incoming task workloads are stored in a queue until they are processed by the computing resource. If there are no waiting or arriving tasks in $[s, t)$, then the available resource $R(s, t)$ is wasted. Otherwise, it is used to process incoming and waiting tasks. For example, a component can process for $R(s, t)$ time units in time interval $[s, t)$.

According to the above explanation, the processing semantics is work conserving, i.e., the processing component has to process available tasks if it has resources available. There are no further assumptions on the scheduling (queuing) discipline, i.e., it may be preemptive, non-preemptive, EDF, fixed priority, or any combination thereof. It can easily be verified, that the above requirement of work conservation leads to the following accumulated processing time $Q(s, t)$ in interval $[s, t)$, i.e., the accumulated time a component is spending to operate on incoming (and queued) workload

$$Q(s, t) = \inf_{s \leq u \leq t} \{R(s, t) - R(s, u) + W(s, u)\} \tag{5.1}$$

where we suppose that at time s there are no buffered tasks, see also [LBT01a, TCN00].

We are interested in determining the upper bound on the temperature under any possible workload that is bounded by some event stream characteristics, e.g., periodic with arbitrary initial phase, periodic with jitter, sporadic, or bursty. To this end, we suppose that the cumulative workload W is upper bounded using the concept of an arrival curve

$$W(s,t) \leq \alpha(t-s) \qquad \forall s < t \tag{5.2}$$

where $\alpha(0) = 0$, see for example [LBT01a]. A tight upper arrival curve is a monotonically increasing staircase function, i.e., it has slope 0 almost everywhere. Furthermore, it is sub-additive, i.e., it satisfies $\alpha(a) + \alpha(b) \geq \alpha(a+b)$ for all $a, b \geq 0$. Note that in case of several independent workload functions W_i that are bounded individually by arrival curves α_i and that need to be concurrently processed in a single component, the accumulated workload can be bounded by

$$W(s,t) \leq \sum_{(i)} \alpha_i(t-s)$$

As a consequence, the results in this chapter will hold for components with several event inputs as well.

In a similar way, the resource availability R can be upper and lower bounded using a pair of upper and lower service curves

$$\beta^l(t-s) \leq R(s,t) \leq \beta^u(t-s) \qquad \forall s < t \tag{5.3}$$

where $\beta^l(0) = \beta^u(0) = 0$.

We can now determine an upper bound on the accumulated computing time according to [LBT01a, WMT06] by using (5.2) and (5.3)

$$Q(t-\Delta, t) \leq \gamma(\Delta) = \min\{(\alpha \otimes \beta^u) \oslash \beta^l, \beta^u\} \tag{5.4}$$

where $(f \otimes g)(\Delta) = \inf_{0 \leq \lambda \leq \Delta}\{f(\Delta - \lambda) + g(\lambda)\}$ and $(f \oslash g)(\Delta) = \sup_{\lambda \geq 0}\{f(\Delta + \lambda) - g(\lambda)\}$.

Narratively speaking, the maximum amount of the workload processed during the time interval Δ is bounded by $\gamma(\Delta)$. When there is more workload available than the provided resource during the interval, it is bounded by the maximum available resource ($\min\{..., \beta^u\}$). If not, $\gamma(\Delta)$ is determined by considering the combination of workload and resource ($(\alpha \otimes \beta^u) \oslash \beta^l$). The accumulated workload (that is not processed previously) does affect γ and this is bounded by the minimum resource availability (β^l) This is also the motivation for assuming a work conserving semantics in this chapter. Without this work-conserving assumption, arbitrarily many workload can be delayed and accumulated regardless of resource availability making γ unbounded. In case of full resource availability, this γ is known to be $\alpha \otimes \beta^u$ according to the Real-Time and Network Calculus[LBT01a, WMT06].

Because of (5.1) and (5.4), the accumulated computing time $Q(s,t)$ for any fixed s as well as its upper bound $\gamma(t-s)$ are monotonically increasing. We now can define the rate function $S(t)$ which represents the rate (in workload per time unit) by which the computing resource is processing:

$$S(t) = \frac{dQ(s,t)}{dt} \tag{5.5}$$

If the computing resource is always fully available, then we find $R(s,t) = \beta^u(t-s) = \beta^l(t-s) = t-s$ as in any time interval of length $t-s$ the computing resource can fully operate on available tasks. In this case, $Q(s,t)$ always has either slope 1 or 0. In other words, the rate function satisfies $S(t) \in \{0,1\}$, i.e., $S(t)=1$ and $S(t)=0$ denote that the processing component is in 'active' and 'idle' mode at time t, respectively. [2] However, such a simplified 'active/idle' mode classification is no longer valid if we allow for general resource availabilities. In this case, the rate function can take any value $0 \leq S(t) \leq 1$.

Now, we need to characterize the mapping of the operation modes to the corresponding power consumption.

5.4.2 Power Models

A well accepted model for the frequency and voltage dependency of the dynamic power consumption P is $P \propto v^{\varrho} f$ where v denotes the supply voltage, $\varrho \geq 2$ models the superlinear dependence of power on the given supply voltage, and f denotes operating frequency in CMOS circuits, respectively. That is, the dynamic power consumption of a single operation mode is determined by the combination of v and f which may be chosen dynamically at run-time in modern processor architectures. The processor frequency f is proportional to the execution rate $S(t)$.

For the sake of simplicity, the supply voltage v is assumed to be constant in the following discussions and therefore, the dynamic power P depends linearly on f which is proportional to the rate function $S(t)$.

In addition, we may model the temperature dependence of leakage power by means of a linear approximation [YCTK10b, LDSY07], which finally yields

$$P(t) = \phi T + \rho S(t) + \psi \tag{5.6}$$

Where $\phi T + \psi$ and ρ are the leakage and dynamic power, respectively. That is, ϕ is a temperature-dependent coefficient for the leakage, while the dynamic power in the active status is ρ.

We also will consider a different, well accepted model which applies to the case where $S(t) \in \{0,1\}$. If $S(t)=0$, then the processor is in 'idle' mode

---

[2] As $S(t)$ implies an operating mode at moment t, it is not a continuous function.

with power P^i, if $S(t) = 1$, then the processor is in the 'active' mode with power P^a:

$$P(t) = \begin{cases} P^i & \text{if } S(t) = 0 \\ P^a & \text{if } S(t) = 1 \end{cases} \quad \text{with} \quad P^i = \phi^i T + \psi^i, \quad P^a = \phi^a T + \psi^a \tag{5.7}$$

5.4.3 Thermal Models

In order to show the versatility of our approach, we will consider three well accepted power-temperature models based on various forms of linear and quadratic differential equations, see also [MMA+07, WB06a, WB06b, WB08, ZC07].

The first two models we consider are based on the following differential equation

$$C\frac{dT}{dt} = P - G \cdot (T - T_{amb}) \tag{5.8}$$

where C, P, G and T_{amb} denote the thermal capacity, the generated power, the thermal conductance, and the ambient temperature, respectively.

Active/Idle Model:

In this case, the system is either in active or idle mode based on the rate $S(t)$, i.e., $P(t) \in \{P^i, P^a\}$, see (5.7).

The steady-state temperatures with rates $S(t) = 0$ and $S(t) = 1$ can be determined as

$$T_0^\infty = \frac{GT_{amb} + \psi^i}{G - \phi^i} \quad , \quad T_1^\infty = \frac{GT_{amb} + \psi^a}{G - \phi^a}$$

A closed-form solution of (5.8) yields

$$T(t) = T^\infty + (T(t_0) - T^\infty) \cdot e^{-\frac{G-\phi}{C}\cdot(t-t_0)} \tag{5.9}$$

as long as the system is in a constant mode (active or idle) for $t \geq t_0$ and T^∞ is the corresponding steady-state temperature.

Continuous Mode Model:

Considering the continuous power model (5.6), we derive from (5.8) the differential equation

$$C\frac{dT}{dt} = (\phi T + \rho S(t) + \psi) - G(T - T_{amb}) \tag{5.10}$$

In addition, we consider an additional temperature-dependent parameter, namely the thermal conductance of silicon. It is reported that the thermal resistance (reciprocal of conductance) can be linearly approximated

[WSMM01a], thus the temperature-dependent conductance can be modeled as follows:

$$G(T) = \frac{1}{R_0 + R_1 T} \tag{5.11}$$

Combining (5.10) and (5.11), we derive the following differential equation

$$\frac{dT}{dt} = \frac{T_{amb} - T}{L_0 + L_1 T} + MT + N \tag{5.12}$$

where $L_0 = CR_0$, $L_1 = CR_1$, $M = \phi/C$, and $N = (\rho S(t) + \psi)/C$.

Linear Model:

Often, the physical computing device contains several interacting thermal layers such as silicon and copper. In this case, linear models based on matrix linear differential equations are well established, see e.g., [SSS$^+$04b, HSG$^+$09]. Based on the physical structure of the device, one can derive the corresponding impulse response function $h(t)$ which leads to the closed-form solution

$$T(t) = T_0^\infty + \int_0^t S(u) \cdot h(t-u) du \tag{5.13}$$

where T_0^∞ denotes the steady-state temperature at constant rate $S = 0$. Note again, that $S(t)$ could be replaced in some monotonic power function $P(S(t))$.

5.4.4 Soundness of Models

An *active/idle model* is called *proper* if it satisfies the following two properties:

- In order to guarantee a stable thermal model, we require that $G > \phi^i$ and $G > \phi^a$.
- We also require that the steady-state temperature in the active mode is larger than that in the idle mode, i.e., $\frac{GT_{amb}+\psi^i}{G-\phi^i} < \frac{GT_{amb}+\psi^a}{G-\phi^a}$.

We call a *continuous mode model* proper, if the temperature converges to a certain value for constant execution rate $S(t)$. We can determine potential steady-state temperatures under constant rate $S(t)$ by setting $\frac{dT}{dt} = 0$ in (5.12) as

$$\frac{1}{2L_1 M}\left(1 - L_0 M - L_1 N \pm \sqrt{(L_0 M + L_1 N - 1)^2 - 4L_1 M(L_0 N + T_{amb})}\right)$$

As a consequence, the following condition needs to hold for a *proper continuous mode model*:

$$(L_0 M + L_1 N - 1)^2 \geq 4L_1 M(L_0 N + T_{amb}) \tag{5.14}$$

More precise insight on the $proper$ continuous mode model is given in Figure 5.3. From the two real potential steady-state temperatures only the lower one is stable. Therefore, we can conclude that the steady-state temperature for a constant rate $S(t)$ is

$$T^{\infty} = \frac{1}{2L_1M}\left(1 - L_0M - L_1N - \sqrt{(L_0M + L_1N - 1)^2 - 4L_1M(L_0N + T_{amb})}\right). \tag{5.15}$$

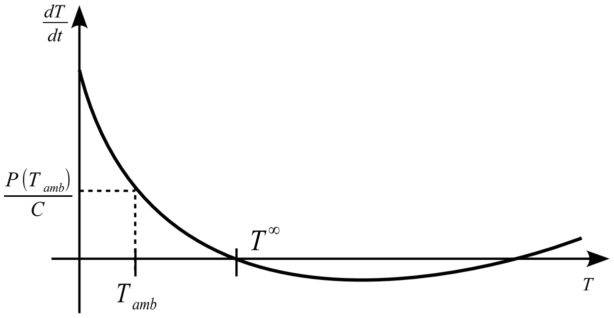


Figure 5.3: Illustration of $\frac{dT}{dt}$ in (5.12) for a constant operation mode.

Finally, we call a *linear model* proper, if the corresponding impulse response function $h(t)$ is strictly monotonically decreasing, i.e., $h(s) > h(t)$ for all $s > t$.

5.4.5 Problem Definition

Now, we can formulate the worst-case peak temperature analysis problem:

> *Given is a work-conserving component characterized by a proper thermal model. The objective is to determine the peak temperature T^* for any cumulative workload W that complies to a given sub-additive arrival curve α and any resource availability R bounded by the wide-sense increasing service curves β^u and β^l, with $\beta^u(0) = \beta^l(0) = 0$.*

The requirement that the arrival curve be sub-additive ensures tightness of bounds, and the results obtained are also tight, see [LBT01b]. Specifically, we say that an arrival curve α is sub-additive if it satisfies (5.16):

$$\forall \delta_1, \delta_2 \geq 0 : \alpha(\delta_1) + \alpha(\delta_2) \geq \alpha(\delta_1 + \delta_2), \tag{5.16}$$

The most naive solution to this problem is to state that T^* is upper-bounded by T_1^{∞} in a fully-utilized mode with $S(t) = 1$, which simply ignores the arrival/service curve by assuming the component is always fully utilized and the computing resource is fully available. However, this is far beyond acceptable when the utilization is low. Therefore, we would like to determine a tight upper bound on T^*, and if possible, even determine a workload trace W and a resource availability R that leads to the peak temperature T^*.

5.5 Thermal Analysis

In order to determine such an upper bound on the peak temperature T^*, we will at first show how to construct a worst case computing time sequence. This result will then be used to determine the desired upper bound. Finally, we will discuss the tightness of this bound and computational aspects.

5.5.1 Worst-Case Computing Time

As a main prerequisite for constructing the peak temperature, we need two properties of the underlying power and thermal models, namely *monotonicity* and *shift*. These two properties will be shown based on the generic form of the thermal differential equation

$$\frac{dT}{dt} = H(S, T) \tag{5.17}$$

where the dependence of $H(S, T)$ on the execution rate S reflects the time-varying power consumption. In other words, H already incorporates the power generation sequence.

We will now show that the generic thermal models as defined above satisfy the thermal *monotonicity* property.

Lem. 5.1. *(Monotonicity) Suppose we consider two solutions of (5.17) with different initial temperatures T_0 at time s. Then the solution $T(t)$ with the higher initial temperature will at no time $t \geq s$ be smaller than the solution with lower initial temperature.*

Proof. Let us suppose that the above theorem is false. Then the temperature trace with the higher initial temperature at s has the lower temperature at some time t. As a consequence, the two temperature traces cross in between, i.e. there exists a time $s < t_0 \leq t$ where the temperatures of the two sequences are equal. As the two sequences have equal derivative $H(S, T)$ for equal temperatures, their temperature at t will be equal, which contradicts the assumption.
□

In terms of the various thermal models that have been described in the previous section, one can interpret the above lemma as follows: If the power traces are the same, then a higher initial temperature does not lead to a smaller final temperature.

As the next prerequisite in constructing an upper bound on the temperature, we will show in lemma 5.2 under what conditions a time shift of the rate function S leads to temperature increase.

Lem. 5.2. *(Shifting) Given is a differential equation of the form (5.17). We consider two different execution rate traces in the time interval* $[0, 2\delta)$, *namely*

$$S(t) = \begin{cases} S_1 & \textit{if } 0 \le t < \delta \\ S_2 & \textit{if } \delta \le t < 2\delta \end{cases}$$

$$\overline{S}(t) = \begin{cases} S_1 - \sigma & \textit{if } 0 \le t < \delta \\ S_2 + \sigma & \textit{if } \delta \le t < 2\delta \end{cases}$$

for some rate shift $\sigma \ge 0$. *Then the corresponding temperature traces* $T(t)$ *and* $\overline{T}(t)$ *satisfy*

$$\overline{T}(2\delta) > T(2\delta)$$

for $T(0) = \overline{T}(0) = T_0$ *and* $\delta \to 0$ *if either*

$$H(S_1 - \sigma, T_0) + H(S_2 + \sigma, T_0) > H(S_1, T_0) + H(S_2, T_0) \tag{5.18}$$

or

$$\begin{aligned} H(S_1 - \sigma, T_0) + H(S_2 + \sigma, T_0) = H(S_1, T_0) + H(S_2, T_0) \quad \wedge \\ H(S_1 - \sigma, T_0) \cdot \frac{\partial H(S_2 + \sigma, T)}{\partial T}(T_0) > H(S_1, T_0) \cdot \frac{\partial H(S_2, T)}{\partial T}(T_0) \end{aligned} \tag{5.19}$$

Proof. Neglecting higher order terms in δ, we can write

$$\begin{aligned} T(2\delta) &= T_0 + \delta[H(S_1, T_0) + H(S_2, T_0 + \delta H(S_1, T_0))] \\ \overline{T}(2\delta) &= T_0 + \delta[H(S_1 - \sigma, T_0) + H(S_2 + \sigma, T_0 + \delta H(S_1 - \sigma, T_0))] \end{aligned}$$

The condition $\overline{T}(2\delta) > T(2\delta)$ is satisfied if

$$H(S_1 - \sigma, T_0) + H(S_2 + \sigma, T_0 + \delta H(S_1 - \sigma, T_0)) > H(S_1, T_0) + H(S_2, T_0 + \delta H(S_1, T_0))$$

This condition directly leads to (5.18,5.19) if we use the following relations

$$\begin{aligned} H(S_2, T_0 + \delta H(S_1, T_0)) &= H(S_2, T_0) + \delta H(S_1, T_0) \frac{\partial H(S_2, T)}{\partial T}(T_0) \\ H(S_2 + \sigma, T_0 + \delta H(S_1 - \sigma, T_0) &= H(S_2 + \sigma, T_0) + \delta H(S_1 - \sigma, T_0) \frac{\partial H(S_2 + \sigma, T)}{\partial T}(T_0) \end{aligned}$$

which hold for $\delta \to 0$.

□

Based on this lemma, we will show later that the temperature at some measuring time gets larger if we 'shift' some 'execution rate' σ towards it. The lemma itself uses two execution rate traces $S(t)$ and $\overline{S}(t)$. These two cases correspond to the above mentioned 'shift' towards larger time instances. If (5.18) and (5.19) are satisfied, then we have a higher temperature at some measuring time after we shift the execution rate σ closer to it. We now show that the shift lemma holds for the simple active/idle mode.

Lem. 5.3. *Suppose that we consider the active/idle model according to (5.7), (5.8) with*

$$H(0,T) = P^i - G \cdot (T - T_{amb}) \quad , \quad H(1,T) = P^a - G \cdot (T - T_{amb})$$

Then with $S_1 = 1$*,* $S_2 = 0$ *and* $\sigma = 1$*, the condition (5.19) in lemma 5.2 is satisfied if the model is proper.*

Proof. Condition (5.19) is equivalent to

$$\begin{gathered} H(0,T_0) + H(1,T_0) = H(1,T_0) + H(0,T_0) \quad \wedge \\ H(0,T_0)\frac{\partial H(1,T)}{\partial T}(T_0) > H(1,T_0)\frac{\partial H(0,T)}{\partial T}(T_0) \end{gathered}$$

which is equivalent to

$$\begin{gathered} (\phi^i T_0 + \psi^i - G \cdot (T_0 - T_{amb}))(G - \phi^a) < (\phi^a T_0 + \psi^a - G \cdot (T_0 - T_{amb}))(G - \phi^i) \Leftrightarrow \\ \frac{\psi^i + GT_{amb}}{G - \phi^i} < \frac{\psi^a + GT_{amb}}{G - \phi^a} \end{gathered}$$

if the model is proper. The last relation is true as the active/idle model is proper.
□

From the above lemma, we can conclude that the shift lemma holds for the active/idle model if it is *proper*. In other words, exchanging active and idle modes such that the active mode gets closer to the measurement time increases the temperature.

Now, we will show a similar condition for the more complex continuous mode model.

Lem. 5.4. *Suppose that we consider the continuous model according to (5.12) with*

$$H(S,T) = \frac{T_{amb} - T}{L_0 + L_1 T} + MT + N(S)$$

Then the condition (5.19) in lemma 5.2 is satisfied if

$$\sigma > 0 \quad \wedge \quad T_0 < T^\infty$$

Proof.

Due to the linearity of $N(S)$, we have $H(S_1 - \sigma, T_0) + H(S_2 + \sigma, T_0) = H(S_1, T_0) + H(S_2, T_0)$. Then we find that (5.19) is equivalent to

$$(\frac{T_{amb} - T}{L_0 + L_1 T} + MT + N(S_1 - \sigma)) \cdot \varsigma > (\frac{T_{amb} - T}{L_0 + L_1 T} + MT + N(S_1)) \cdot \varsigma$$

Where $\varsigma = \frac{-L_0 - L_1 T_{amb}}{(L_0 + L_1 T)^2}$.

If $T_0 < T_{min} = M/2L$ then this condition is equivalent to

$$\frac{T_{amb} - T}{L_0 + L_1 T} + MT + N(S_1 - \sigma) < \frac{T_{amb} - T}{L_0 + L_1 T} + MT + N(S_1) \Leftrightarrow$$
$$N(S_1 - \sigma) < N(S_1) \quad \Leftrightarrow \quad \sigma > 0$$

□

In summary, the above lemma states that one gets a higher temperature if we shift the rate σ towards the measuring time in case of the continuous model.

Finally, we show that the shift-property as defined in lemma 5.2 also holds for the linear model.

Lem. 5.5. *(Shift) Given is a linear temperature model of the form (5.13). We consider two different execution rate traces in the time interval* $[0, 2\delta)$*, namely*

$$S(t) = \begin{cases} S_1 & \textit{if } s \le t < s + \delta \\ S_2 & \textit{if } s + \delta \le t < s + 2\delta \end{cases}$$
$$\overline{S}(t) = \begin{cases} S_1 - \sigma & \textit{if } s \le t < s + \delta \\ S_2 + \sigma & \textit{if } s + \delta \le t < s + 2\delta \end{cases}$$

for some rate shift $\sigma > 0$*. It the model is proper, then the corresponding temperature traces* $T(t)$ *and* $\overline{T}(t)$ *satisfy*

$$\overline{T}(s + 2\delta) > T(s + 2\delta)$$

for $T(s) = \overline{T}(s)$ *and* $\delta \to 0$*.*

Proof. From (5.13) we can derive by simple algebraic transformations

$$\overline{T}(s + 2\delta) - T(s + 2\delta) = \sigma \left(\int_s^{s+\delta} h(t-u)du - \int_{s+\delta}^{s+2\delta} h(t-u)du \right)$$

Therefore, $\overline{T}(s + 2\delta) > T(s + 2\delta)$ is equivalent to

$$\int_0^{\delta} h((t-s) - u)du > \int_0^{\delta} h((t-s) - \delta - u)du$$

which is satisfied if $h(t) > h(t - \delta)$ for all δ, i.e., if the model is proper.
□

The next lemma shows for all considered power and temperature models, that we obtain a higher temperature at some time τ if in *any* interval ending at τ the component has larger accumulated computing time. This lemma provides the foundation for the main theorem of this Section.

Lem. 5.6. *(Worst-case Computing Time) Given is a proper thermal model, i.e., a model that satisfies the shift condition as defined in lemmata 5.2 and 5.5, as well as some time instance* τ*. In addition, we consider two accumulated computing time functions* Q *and* $\overline{Q}$ *which satisfy*

$$\overline{Q}(\tau - \Delta, \tau) \geq Q(\tau - \Delta, \tau)$$

for all $0 \leq \Delta \leq \tau$*. Then, if* $\overline{T}(0) = T(0)$ *we have* $\overline{T}(\tau) \geq T(\tau)$*, i.e., the temperature at time* τ *is not higher if we use the accumulated computing time function* Q *instead of* $\overline{Q}$*.*

Proof. First note that because of (5.5), the condition of the lemma translates equivalently to the following condition on the corresponding rate functions S and $\overline{S}$:

$$\int_{\tau-\Delta}^{\tau} \overline{S}(t)\, dt \geq \int_{\tau-\Delta}^{\tau} S(t)\, dt$$

The following algorithm performs a stepwise transformation of $S(t)$ into $\overline{S}(t)$ using the elementary rate shift operations in lemmata 5.2 and 5.5. As a result, we can show that in each step the temperature will increase. In order to simplify the proof technicalities, we suppose discrete time, i.e., $S(t)$ and $\overline{S}(t)$ may change values only at multiples of δ. In other words, $S(t)$ and $\overline{S}(t)$ are constant for $t \in [k\delta, (k+1)\delta)$ for all $k \geq 0$. Let us define $\tau = k_{\text{max}}\delta$. We now execute the following algorithm:

1. Determine the smallest $1 \leq k_1 \leq k_{\text{max}}$ such that $S(\tau - k_1\delta) < \overline{S}(\tau - k_1\delta)$. If there is no such k_1, then $S(t) = \overline{S}(t)$ for all $0 \leq t \leq \tau$ and therefore, $\overline{T}(\tau) = T(\tau)$ and the algorithm stops.
2. Determine the smallest k_2 with $k_1 < k_2 \leq k_{\text{max}}$ such that $S(\tau - k_2\delta) \neq 0$. In case such a k_2 does not exist, then $\overline{T}(\tau) \geq T(\tau)$ holds and the algorithm stops. Otherwise, execute the following steps:
   (a) Set $\sigma = \min\{S(\tau - k_2\delta), \overline{S}(\tau - k_1\delta) - S(\tau - k_1\delta)\}$.
   (b) For all i starting from k_2 down to $k_1 + 1$ change $S(t)$ as follows: $S(t) := S(t) - \sigma$ for $t \in [\tau - i\delta, \tau - (i-1)\delta)$ and $S(t) := S(t) + \sigma$ for $t \in [\tau - (i-1)\delta, \tau - (i-2)\delta)$.
3. If $S(\tau - k_1\delta) < \overline{S}(\tau - k_1\delta)$ is still true, continue with step 2. Otherwise, go to step 1.

Note that σ is always positive in step 2(a) as required in lemmas 5.3, 5.4 and 5.5. Now, we can simply prove using lemma 5.2 that after each execution of step 2, $T(\tau)$ decreases until it reaches $\overline{T}(\tau)$. Therefore, the initial $T(\tau)$ was not larger than $\overline{T}(\tau)$.

□

Based on the above lemma 5.6 we will show the first main result of this Section. The following theorem 5.1 provides a constructive method to determine the worst-case accumulated computing time Q^* for a work-conserving component.

Thm. 5.1. *(Worst-case Temperature) Given a work-conserving processing component with the computational model (5.1), one of the power models defined in Section 5.4.2, and a thermal model as described in Section 5.4.3. The thermal model is supposed to be proper according to Section 5.4.4. Then the following holds:*

- *Suppose that the accumulated computing time function* $Q^*(0,\Delta) = \gamma(\tau) - \gamma(\tau-\Delta)$ *for all* $0 \leq \Delta \leq \tau$ *leads to temperature* $T^*(\tau)$ *at time* τ*, where* γ *is defined in (5.4). Then* $T^*(\tau)$ *is an upper bound on the highest temperature* $T(\tau)$ *for all feasible workload traces that are bounded by the arrival curve* α *and service curve* β *according to (5.2) and (5.3) respectively.*
- *If in addition* $T(0) \leq T_0^\infty$ *holds, where* T_0^∞ *is a steady-state temperature for the constant rate function* $S(t) = 0$*, then for any feasible workload trace we find* $T^*(\tau) \geq T(t)$ *for all* $0 \leq t \leq \tau$*.*

Proof. At first, we show that $Q^*(0,\Delta) = \gamma(\tau) - \gamma(\tau-\Delta)$ satisfies (5.4). We have $Q^*(t-\Delta,t) = Q^*(0,t) - Q^*(0,t-\Delta) = \gamma(\tau) - \gamma(\tau-t) - \gamma(\tau) + \gamma(\tau-t+\Delta) = \gamma(\tau-t+\Delta) - \gamma(\tau-t)$. Therefore, we have to show that $\gamma(\tau-t+\Delta) - \gamma(\tau-t) \leq \gamma(\Delta)$ for all $0 \leq t \leq \tau$ which is satisfied if $\gamma(a+b) \leq \gamma(a) + \gamma(b)$ for all $a, b \geq 0$.

It is well known from network and real-time calculus that tight upper arrival/service curves are sub-additive and tight lower service curves are superadditive, i.e., they satisfy $\alpha(a)+\alpha(b) \geq \alpha(a+b)$, $\beta^u(a)+\beta^u(b) \geq \beta^u(a+b)$ and $\beta^l(a)+\beta^l(b) \leq \beta^l(a+b)$, see also [TCN00]. Using these properties in (5.4) results in $\gamma(a) + \gamma(b) \leq \gamma(a+b)$.

Now, we will show the first item of the theorem by contradiction. Suppose that there is an accumulated computing time Q which leads to a higher temperature $T(\tau)$ at time τ. Then according to lemma 5.6 there exists some $\Delta \leq \tau$ such that $Q^*(\tau-\Delta,\tau) < Q(\tau-\Delta,\tau)$. As we know that $Q^*(\tau-\Delta,\tau) = \gamma(\Delta) - \gamma(0) = \gamma(\Delta)$ we can conclude that $Q(\tau-\Delta,\tau) > \gamma(\Delta)$. Such a computing time function Q would violate (5.4).

Now, let us prove the second item of the theorem by contradiction. To this end, we denote as $T(t)$ the temperature caused by some accumulated computing time Q. Suppose now that there exists some time $\sigma \leq \tau$ where we have $T(\sigma) > T^*(\tau)$. For an upper bound on $T(\sigma)$, e.g., $T^*(\sigma) \geq T(\sigma)$, we also would find $T^*(\sigma) > T^*(\tau)$. Suppose that we construct such an upper bound using the first item in the theorem, i.e., we choose as a worst case accumulated computing time $Q(\sigma-\Delta,\sigma) = \gamma(\Delta)$ for $0 \leq \Delta \leq \sigma$.

As $Q^*(\tau-\Delta,\tau) = \gamma(\Delta)$, we can now conclude that Q^* shifted by $\tau-\sigma$ and Q are equal, i.e., we have $Q(\sigma-\Delta,\sigma) = Q^*(\tau-\Delta,\tau)$ for $0 \leq \Delta \leq \sigma$. Because of the monotonicity of the thermal model (see lemma 5.1), the assumption $T^*(\sigma) > T^*(\tau)$ would require that the initial temperature $T(0)$ used for the worst case computing time Q is larger then the temperature at time $\tau-\sigma$ when using computing time Q^*.

As the thermal models are supposed to be proper, temperatures in all scenarios, i.e., with all computing time functions, are always larger or

equal than the minimum of the initial temperature and T_0^∞. As we have $T(0) \leq T_0^\infty$, temperatures are always larger or equal $T(0)$. This contradicts the above requirement that the initial temperature $T(0)$ is larger than some temperature that occurs using Q^*.
$\square$

As a result of the above theorem, we can now describe a method to determine an upper bound on the component temperature $T^*(\tau)$ at some time $t = \tau$:

- We start with a given bound on the cumulative workload, i.e., the arrival curve α, and with given upper and lower bounds on the resource availability β^u and β^l, see (5.2) and (5.3).
- We determine the upper bound on the accumulated computing time γ based on (5.4). Using γ, we can determine the worst case computing time function Q^* according to theorem 5.1 and the corresponding rate function S^* in (5.5).
- Corresponding to the chosen power and temperature model, we solve the temperature equation (5.8), (5.10), or (5.13). The model has to satisfy the conditions of the corresponding lemma 5.3, 5.4 or 5.5, i.e., the models should be proper according to Section 5.4.4. The solution may be done analytically or numerically with an appropriate initial temperature $T(0)$ according to theorem 5.1. The temperature at $t = \tau$ is $T^*(\tau)$.

There are still two questions that need to be answered: Under what conditions is the above bound tight? What is a reasonable time τ such that the upper bound $T^*(\tau)$ holds for arbitrary long runs of the system, i.e., $T(t) \leq T^*(\tau)$ for all $t \geq 0$? The next two Sections will answer these questions.

5.5.2 Tightness

Note that theorem 5.1 only provides an upper bound $T^*(\tau)$ on the actual worst-case temperature. In other words, there may be no *single* trace that leads to the critical accumulated computing time $Q^*(0, \Delta) = \gamma(\tau) - \gamma(\tau - \Delta)$. Now, we will show that in the case of maximal resource availability there exists a single trace $W^*(0, \Delta)$ for $0 \leq \Delta \leq \tau$ which (a) is compatible to the given arrival curve α and (b) results in the worst case accumulated computing time $Q^*(0, \Delta)$. Maximal resource availability results in $R(s, t) = t - s$, i.e., in any time interval of length $t - s$ the computing resource is fully available. As a result we find $\beta^u(\Delta) = \beta^l(\Delta) = \Delta$.

We first determine a continuous accumulated workload function $W^*(0, \Delta)$, i.e., which has slopes 1 and 0. It can be interpreted as the limit case of task arrivals with infinitesimally small inter-arrival times and infinitesimally small computation times.

Thm. 5.2. *(Tightness) Suppose that the assumptions from theorem 5.1 hold and the resource availability satisfies $R(s,t) = t - s$ for all $t \geq s$. Then, the continuous workload function $W^*(0,\Delta) = Q^*(0,\Delta)$ for $0 \leq \Delta \leq \tau$*

- *leads to the accumulated computing time $Q^*(0,\Delta)$ according to the computational model (5.1),*
- *complies to the arrival curve α according to (5.2), and*
- *leads to the highest possible temperature $T^*(\tau) \geq T(\tau)$ for any feasible workload trace.*

Proof. With the condition $\beta^l(\Delta) = \beta^u(\Delta) = \Delta$, the computational model in (5.1,5.4) can be simplified to

$$Q(s,t) = \inf_{s \leq u \leq t} \{(t-u) + \alpha(s,u)\} \tag{5.20}$$

$$Q(t-\Delta,t) \leq \gamma(\Delta) = \inf_{0 \leq u \leq \Delta} \{(\Delta - u) + \alpha(u)\} \tag{5.21}$$

Thus, for the first item, we actually need to prove that $Q^*(0,\Delta) = \inf_{0 \leq u \leq \Delta}\{(\Delta - u) + Q^*(0,u)\}$ as $W^*(0,\Delta) = Q^*(0,\Delta)$. At first, we find that there exists a u' such that $(\Delta - u') + Q^*(0,u') = Q^*(0,\Delta)$, namely $u' = \Delta$. Therefore, we only have to show that $(\Delta - u) + Q^*(0,u) \geq Q^*(0,\Delta)$ for all $0 \leq u \leq \Delta$. This condition is equivalent to $(\Delta - u) \geq Q^*(0,\Delta) - Q^*(0,u) = Q^*(u,\Delta)$. As the accumulated processing time in interval $[u,\Delta)$ can not exceed the available service $\Delta - u$, the first item is proven.

With $W^*(0,\Delta) = Q^*(0,\Delta)$, $Q^*(0,\Delta) = \gamma(\tau) - \gamma(\tau - \Delta)$ and $\gamma(\Delta) = \inf_{0 \leq \lambda \leq \Delta}\{(\Delta - \lambda) + \alpha(\lambda)\}$ we find

$$\begin{aligned} W^*(a,b) &= \gamma(\tau - a) - \gamma(\tau - b) \\ &= \inf_{0 \leq \lambda \leq t-a}\{(t - a - \lambda) + \alpha(\lambda)\} - \inf_{0 \leq \eta \leq t-b}\{(t - a - \eta) + \alpha(\eta)\} \\ &\leq \inf_{0 \leq u \leq (b-a)}\{((b-a) - u) + \alpha(u)\} \leq \alpha(b-a) \end{aligned}$$

where we use the fact that $a \leq b \leq \tau$, $\eta \leq \gamma$ as well as the subadditivity of α.

The third item is a simple consequence of theorem 5.1 as (a) W^* leads to the accumulated computing time function Q^* and (b) Q^* leads to the highest temperature $T^*(\tau) \geq T(\tau)$.

□

As has been mentioned above, $W^*(0,\Delta)$ has slope 1 or 0 and corresponds to a continuous arrival of tasks. There are many possibilities to convert such a workload trace into one that has discrete task arrivals, which is compliant to the provided arrival curve α and which leads to the worst-case temperature. In the following, let us describe one of these possibilities.

Lem. 5.7. *(Worst-Case Workload) Let us suppose that the conditions of theorem 5.2 hold. Furthermore, let us suppose that for some constant c the given arrival curve α satisfies $\alpha(\Delta) = c \cdot \lceil \frac{1}{c}\alpha(\Delta)\rceil$ for all $\Delta \geq 0$, i.e., the step size of $\alpha(\Delta)$ is an integer multiple of c. Suppose that the observation time τ is chosen such that $\gamma(\tau)$ according to (5.4) is a multiple of c as well. Then the worst-case accumulated workload $\hat{W}^*(0,\Delta) = c \cdot \lceil \frac{1}{c} W^*(0,\Delta)\rceil$*

- *is piecewise constant with a step size which is an integer multiple of c,*
- *complies to the arrival curve α according to (5.2) and*
- *leads to the highest possible temperature $T^*(\tau) \geq T(t)$ for all $0 \leq t \leq \tau$ for any feasible workload trace.*

Proof. Let us first suppose without restricting the generality that $c = 1$. The first item is obvious from $\hat{W}^*(0,\Delta) = \lceil W^*(0,\Delta)\rceil$.

The second item can be shown as $\hat{R}^*(a,b) = \lceil R^*(0,b)\rceil - \lceil R^*(0,a)\rceil \leq \lceil R^*(0,b) - R^*(0,a)\rceil = \lceil R^*(a,b)\rceil \leq \lceil \alpha(b-a)\rceil = \alpha(b-a)$ for $a < b$.

In order to show the third item, we start from $Q^*(0,\Delta) = \gamma(\tau) - \gamma(\tau - \Delta)$ in theorem 5.1 and $\gamma(\Delta) = \inf_{0 \leq \lambda \leq \Delta}\{(\Delta - \lambda) + \alpha(\lambda)\}$ from (5.21). From the last equation one can observe that $\gamma(\Delta)$ has slope 1 or 0 and it has an integer value if it has slope 0. Therefore, if $\gamma(\tau)$ is integer as well, then we also find that $Q^*(0,\Delta)$ has slope 1 or 0 and it has an integer value if it has slope 0. If we can show that the accumulated workload function $\hat{W}^*(0,\Delta) = \lceil W^*(0,\Delta)\rceil = \lceil Q^*(0,\Delta)\rceil$ leads to the same worst-case accumulated processing time $Q^*(0,\Delta)$ as $W^*(0,\Delta)$, the theorem would hold. Note that $Q^*(0,\Delta) = \inf_{0 \leq u \leq \Delta}\{(\Delta - u) + Q^*(0,u)\}$ from (5.21) in theorem 5.2. Because of the property of $C^*(0,u)$ mentioned before, one can easily deduce that $\inf_{0 \leq u \leq \Delta}\{(\Delta - u) + Q^*(0,u)\} = \inf_{0 \leq u \leq \Delta}\{(\Delta - u) + \lceil Q^*(0,u)\rceil\} = \inf_{0 \leq u \leq \Delta}\{(\Delta - u) + \hat{W}^*(0,u)\}$.
□

Note that $W^*(0,\Delta)$ does not necessarily represent the conventional critical instant scenario that is often used in real-time analysis in order to determine the worst-case timing behavior.

As a result of theorem 5.2 and lemma 5.7, the upper bound $T^*(\tau)$ determined through theorem 5.1 is proven to be tight under the conditions mentioned in theorem 5.2, i.e., there exists a worst case workload trace W^* that actually leads to $T^*(\tau)$ when a system resource is fully available all the time.

5.5.3 Computational Aspects

As has been mentioned already, theorem 5.1 implies a constructive method to determine the upper bound $T^*(\tau)$ for some time τ: Starting from a given arrival curve $\alpha(\Delta)$ for $0 \leq \Delta \leq \tau$ one can determine the function $\gamma(\Delta)$ for $0 \leq \Delta \leq \tau$ using (5.4). With $Q^*(0,\Delta) = \gamma(\tau) - \gamma(\tau - \Delta)$ for all $0 \leq \Delta \leq \tau$ and (5.5) one can determine the critical mode function $S^*(t)$,

$0 \leq t \leq \tau$. It determines the critical distribution of operating modes (5.6) which is used to solve the thermal model, i.e., at time $t = \tau$ we find $T^*(\tau)$.

There is one question remaining: How to choose an appropriate observation time τ such that a bound with an appropriate precision is determined? For simplicity and without restricting the generality, we suppose for the remainder of this section, that the initial temperature satisfies $T(0) = T_0^\infty$, i.e. it equals the steady state temperature of an idle system with constant rate $S = 0$.

As a simple approach, it is possible to determine an upper bound on the precision, given an observation time τ. This does not solve the original question but can be used as a basis for some heuristics, e.g. double the observation time τ until the desired precision is guaranteed. Given τ, we first determine the upper bound $T_0^*(\tau)$ for initial temperature T_0^∞ and the bound $T_1^*(\tau)$ for initial temperature T_1^∞, i.e. the steady state temperature of an active system with constant rate $S = 1$. Due to the monotonicity of the power/temperature mode according to lemma 5.1, we can conclude that $\lim_{t\to\infty} T^*(t) \in [T_0^*(\tau), T_1^*(\tau)]$.

For the active/idle model and the continuous mode model, it is possible to determine the observation time τ explicitly, given a desired precision.

Lem. 5.8. *We are given computational, temperature and power models according to theorem 5.1 and $T(0) = T_0^\infty$, i.e., the steady state temperature for constant rate $S = 0$. Then*

$$\lim_{t\to\infty} T^*(t) \in [T^*(\tau), T^*(\tau) + \Delta T^*]$$

where $T^*(\tau)$ denotes the upper bound determined using theorem 5.1 and observation time τ and where ΔT^* denotes a bound on the corresponding precision. Given ΔT^*, the following expressions provide a lower bound on the observation time τ that achieves this precision:

$$\tau \geq \frac{C}{G - \max\{\phi^i, \phi^a\}} \ln\left(\frac{T_1^\infty - T_0^\infty}{\Delta T^*}\right) \tag{5.22}$$

holds for the active/idle model and

$$\tau \geq \frac{1}{-g} \ln\left(\frac{T_1^\infty - T_0^\infty}{\Delta T^*}\right) \textit{ with } \quad g = \frac{-L_0 - L_1 T_1^\infty + L_1(T_{amb} - T_1^\infty)}{(L_0 + L_1 T_1^\infty)^2} + M \tag{5.23}$$

holds for the continuous mode model where N is computed for $S = 1$.

Proof. At first note, that due to the monotonicity of the power/temperature mode according to lemma 5.1, we find $\lim_{t\to\infty} T^*(t) \in [T_0^*(\tau), T_1^*(\tau)]$. In order to be independent of the actual rate function $S(t)$, we first determine the worst case rate function, i.e., which leads to the worst precision.

Suppose that we integrate (5.17) for a small time step δ and two different initial temperatures T_1 and T_2. With the resulting temperatures T_1' and T_2' we obtain

$$\frac{T_2' - T_1'}{T_2 - T_1} = 1 + \delta \frac{H(S, T_2) - H(S, T_1)}{T_2 - T_1}$$

For the active/idle model, we simply find that this term is $1 - \delta / C \cdot (G - \phi)$ where $\phi \in \{\phi^i, \phi^a\}$. This term is maximal for $\phi = \max\{\phi^i, \phi^a\}$, i.e., the constant rate associated to ϕ. For the continuous mode model we obtain the term $1 + \delta(M - \frac{L_0 + L_1 T_{smb}}{(L_0 + L_1 T_1)(L_0 + L_1 T_1)})$. As larger rates lead to a higher temperature gradient $H(S, T)$, the term is maximal for constant rate $S = 1$.

Now, let us prove (5.22). Using (5.9), we find

$$\Delta T^* = (T_1^\infty - T_0^\infty) e^{\frac{\phi - G}{C} \tau}$$

which directly leads to (5.22).

The proof for the continuous mode model follows. Let $T(T_s, \tau)$ be the resulting temperature at τ with starting temperature T_s for $S = 1$. When the starting temperature is T_1^∞, the temperature stays at T_1^∞. Thus,

$$\Delta T^* = T(T_1^\infty, \tau) - T(T_0^\infty, \tau) = T_1^\infty - T(T_0^\infty, \tau)$$

As it is hard to get the closed form solution for (5.12), we approximate $T(T_0^\infty, \tau)$ with an asymptotic line on dT/dt as shown in Figure 5.4. With $F = \frac{d^2 T}{dt^2}$, the approximated temperature $\tilde{T}$ holds following:

$$\frac{d\tilde{T}}{dt} = F(T_1^\infty)(\tilde{T} - T_1^\infty)$$

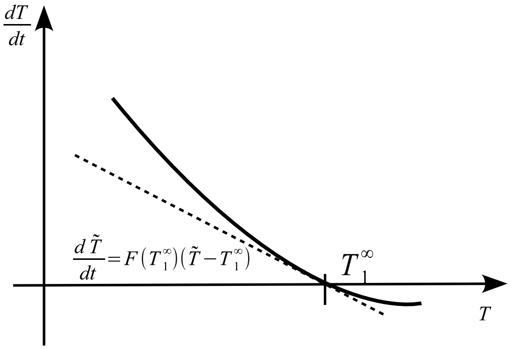


Figure 5.4: Approximated temperature $\tilde{T}$ which has the same stable temperature T_1^∞ but converges slower.

Since $\frac{dT}{dt} \geq \frac{d\tilde{T}}{dt}$ for $T, \tilde{T} \leq T_1^\infty$, $T(T_0^\infty, \tau) \geq \tilde{T}(T_0^\infty, \tau)$, $\tilde{T}$ converges to T_1^∞ slower than T. Using the solution for $\tilde{T}$ in form of (5.9), we have

$$\Delta T^* \geq T_1^\infty - \tilde{T}(T_0^\infty, \tau) = T_1^\infty - (T_1^\infty + (T_0^\infty - T_1^\infty) e^{g\tau})$$

where g is $F(T_1^\infty)$. Simple algebraic transformations with $g = \frac{-L_0 - L_1 T_1^\infty + L_1(T_{amb} - T_1^\infty)}{(L_0 + L_1 T_1^\infty)^2} + M$ lead to (5.23).
$\square$

Following the above lemma, we can determine a suitable observation time τ before the worst-case temperature simulation while guaranteeing a precision on the worst case temperature bound.

5.6 Experimental Analysis

In this section, we will compare our worst-case analysis results with a random simulation of a basic real-time system. For simplicity, we illustrate our techniques assuming a set of processes that are periodic with jitter. Furthermore, we investigate the impact of different task invocation periods and jitter in task arrivals, and observe their relationship to the maximal temperature. Impact of variable resource availability on the temperature is also studied by allowing different service curves.

To ensure repeatability of our experiments and further investigations, our code has been integrated into the MPA-RTC toolbox and is available on-line at `http://www.mpa.ethz.ch/rtctoolbox`.

5.6.1 Benchmarks and Basic Configuration

In addition to the simple example from Section 5.2, a multi-processing video-conferencing system is considered, where three processes are executing on an ARM embedded processor. The system includes a video codec, an audio codec, and a network process which manages the communication, and for illustration it has been configured with a period-jitter-delay model, see [WMT06, WT06b], with parameters summarized in table 5.1.

In the example, the video codec operates in a range varying from 12 frames per second (fps) to 50fps, being able to provide different output video qualities. This offers us the possibility to investigate a large range of invocation periods between 20ms and 90ms. For illustration purposes we set the audio codec to operate at a similar sampling rate, with an invocation period of 20ms, which in the real system means just pre-processing (buffering) audio samples. Finally, the network process will be invoked as well with a period of 20ms. We assumed the deadline of each task identical to its invocation period. For the exact meaning of all parameters in table 5.1, please refer to [WT06b]. System parameters are summarized in table 5.2. Coefficients for the temperature dependent thermal resistance (R_0 and R_1) are taken from [WSMM01b] and properly scaled to have the same thermal conductance as [YCTK10b] at 300K. Power parameters are adapted from [YCTK10a] and scaled down to be proper according to (5.14). In all our experiments we start simulations with the initial temperature $T(0) = T_0^\infty = 319.31$K, calculated from parameters given in table 5.2 and

Table 5.1: Parameters of the video conferencing application.

	Video	Audio	Network
period	[20, 90]ms	30ms	30ms
jitter	[20, 90]ms	10ms	10ms
min. interarrival	1ms	1ms	1ms
execution demand	6ms	3ms	2ms
deadline	[20, 90]ms	30ms	30ms

Table 5.2: Thermal and power parameters of the considered embedded system architecture.

R_0	R_1	C	ϕ	ρ	ψ
$0.052\frac{1}{W}$	$0.0123\frac{1}{WK}$	$0.0218\frac{J}{K}$	$0.07\frac{W}{K}$	$9.8W$	$-17.5W$

the observation time interval $\tau = 1.2$s, see lemma 5.8. For the power mode in equation (5.6) and the continuous thermal model in equation (5.12) have been used for the simulation.

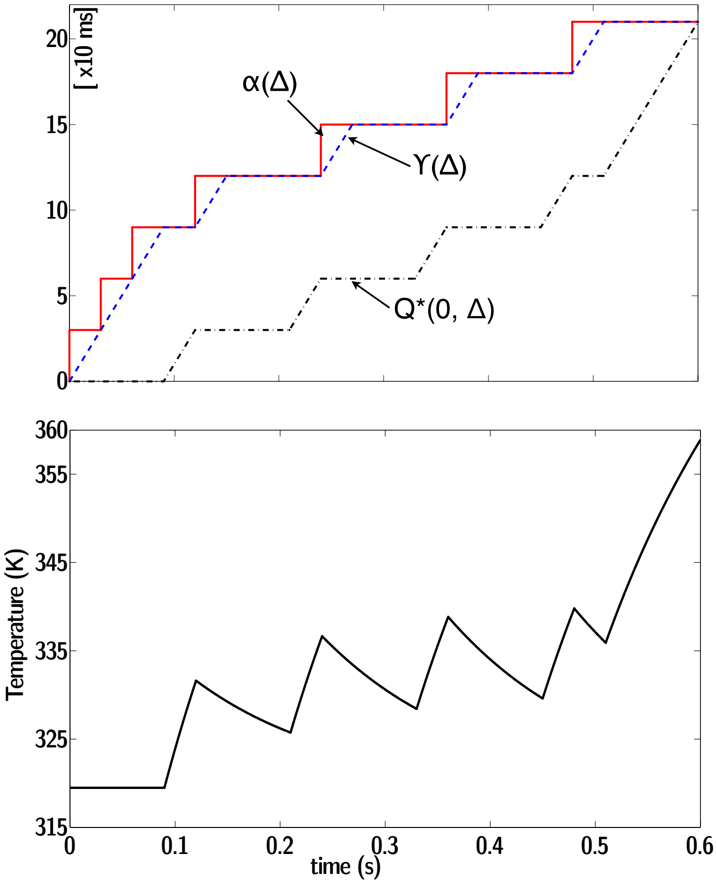


Figure 5.5: Relevant analysis quantities related to the simple example from Section 5.2.

5.6.2 Peak Temperature Analysis

At first, let us look again at the simple example from Section 5.2, i.e., a *single* task stream with period 120ms, jitter 240ms, computation time 30ms and minimal inter-arrival time 30ms on the fully serviced processor. The following Figure 5.5 shows all relevant quantities, i.e., $\alpha(\Delta)$ according to (5.2), $\gamma(\Delta)$ according to (5.4), the worst-case accumulated computing time $Q^*(0,\Delta)$ from theorem 5.1 and the observation interval length $\tau = 0.6$s. It is intuitively shown that the workload is placed as late as possible upto τ causing the bursty shot at the end. A part of the corresponding temperature trace and workload according to lemma 5.7 has already been shown in Figure 5.2, but using the observation interval $\tau = 1.2$s.

The video conferencing example described in table 5.1 is also analyzed in terms of peak temperature. The invocation period of video task is fixed to 50ms in this experiment. For the comparison to a temperature simulation of random traces, we use 100 randomly generated task arrivals that conform to the workload specification. Figure 5.6 shows the correctness of the computed upper bound according to theorem 5.1 and the limited coverage of random simulations. A similar experiment, but using reduced order models derived from the previous chapters is shown in Figure 5.7.

Five different transient temperatures in the interval $[0s, 1.2s)$ are drawn in Figure 5.6: (a) the time critical instance (the workload trace that fits to the critical instant for timing analysis by releasing the workload as early as possible at the beginning), (b) the thermal critical instance with constant G, (c) the thermal critical instance (generated by Theorem 5.1) with variable G of (5.11), (d) the thermal critical instance with varying G and infinitesimally small period/jitter, and (e) 100 randomly generated workload traces. All traces start from the stead-state idle temperature $T_0^\infty = 319.49$K. For (d), we change the period and jitter infinitesimally small while keeping the utilization the same.

The time critical instance has higher transient temperature before its first idle time than other traces. However, its temperature starts to decrease after that moment, and does not lead to the worst-case peak temperature. The highest temperature observed is 350.721K. In contrast, the 100 random simulations might keep the system at higher temperature later on, but still does not capture the worst-case peak temperature. The highest peak temperature is 354.4K for random simulations. As shown in Figure 5.5, the thermal critical trace first warms up the system with periodic arrivals and then heats up the system with burst arrivals and jitters at the end around τ and the resultant worst-case transient temperature 355.652K in Figure 5.6.

The effect of a burst on system temperature is illustrated in Figure 5.6(d). We make the period and jitter value infinitesimally small in (d), while keeping the total utilization of workload as the same as others. This eventually makes the trace equivalent to the average workload removing

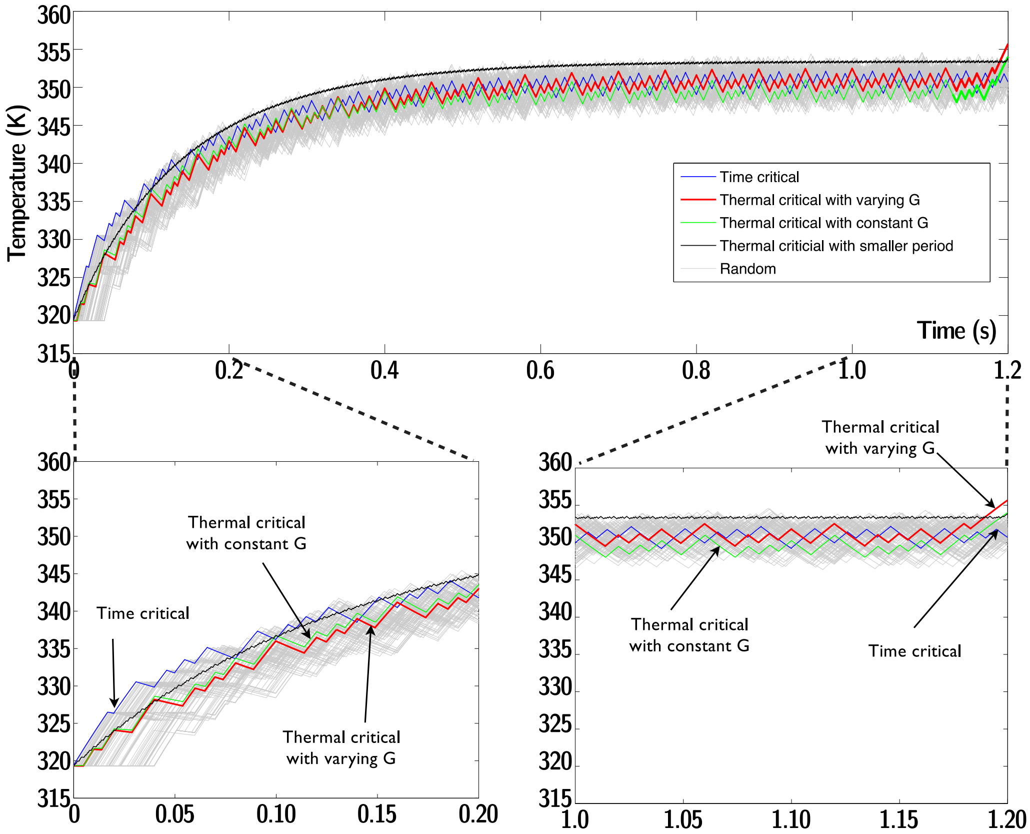


Figure 5.6: Comparison between worst-case temperature estimate and other traces: (a) time critical instance, (b) thermal critical instance with temperature dependent thermal conductivity, (c) thermal critical instance with constant conductivity, (d) thermal critical instance with infinitesimally small period/jitter (varying conductivity), and (e) 100 random simulations.

the bursty shot at the end. This can be seen from the temperature trajectory in Figure 5.6(d) resulting in the worst-case temperature of 353.862K.

The effect of temperature variant thermal conductivity is presented in the difference between (b) and (c). Note that we take the mean value of varying G as a constant value using $G_{const} = \frac{G(T_{amb})+G(T_1^\infty)}{2}$ and (5.10) is used to calculate the transient temperature. In case of constant conductivity, the worst-case temperature is far underestimated as 354.03K.

The worst-case temperature $T^*(\tau)$, however, only gives the peak temperature of a feasible trace. As shown in lemma 5.8, for estimating the worst-case peak temperature T^*, we also need the peak temperature at time τ, by starting at the possible highest temperature $T_{l=1}^\infty$ for the same trace. Table 5.3 demonstrates the temperature bound $[T_0^*(\tau), T_1^*(\tau)]$ by varying τ from 0.3s to 2.0s when task video task period and jitter are both $20ms$. Note that, for different values of τ, the worst-case traces are also different. When τ is small, the bound is not precise. For instance, the bound

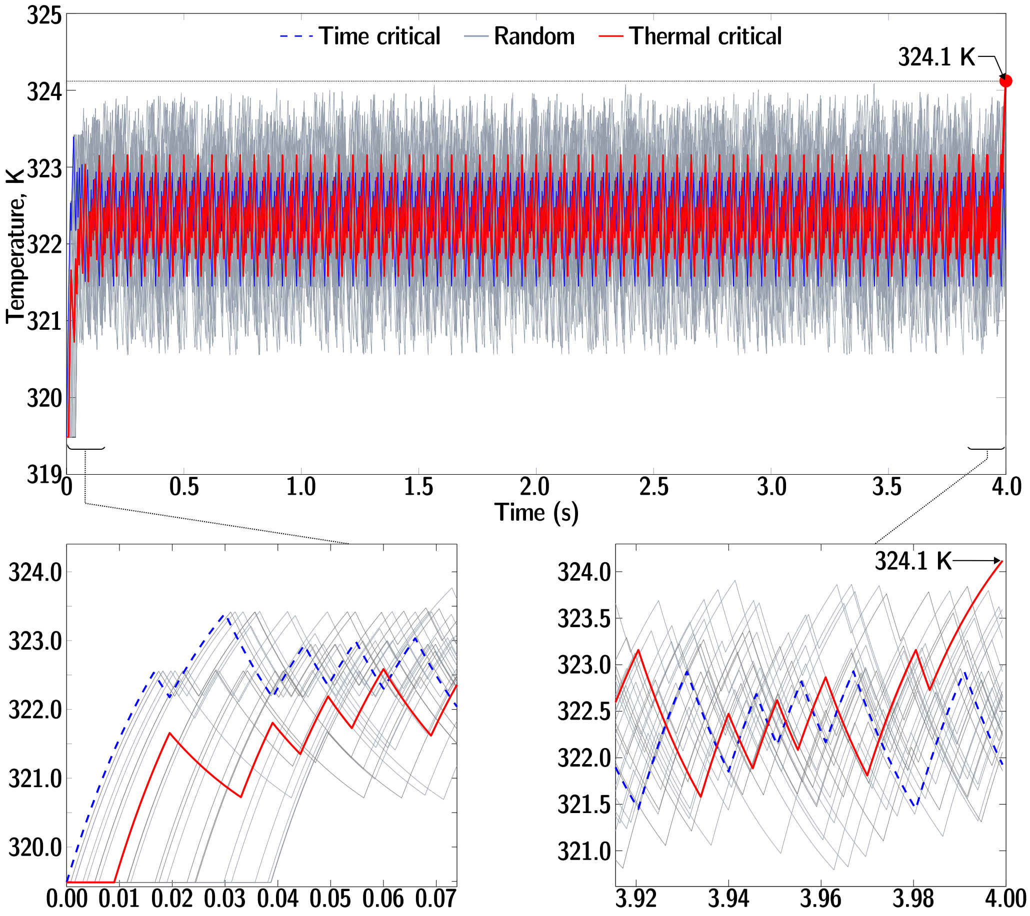


Figure 5.7: An experiment similar to the one shown in Figure 5.6 but using the single pole ADPCM model derived from the previous chapters. The overall results match that of Figure 5.6: a thermal critical instance will lead to the peak worst case temperature.

Table 5.3: Bounds on worst-case peak temperature analysis for different τ values.

τ	0.3s	0.6s	0.9s	1.2s	2.0s
$T_{l=0}(\tau)$	350.794K	354.853K	355.535K	355.652K	355.681K
$T_{l=1}(\tau)$	366.318K	357.573K	356.004K	355.732K	355.681K

is [354.853K,357.573K] in case of $\tau = 0.6$s. However, when τ is getting larger, the bound tends to converge, with the precision of the 4th digit of decimal point, i.e., 355.6810K. As a result, we can conclude that the worst-case peak temperature is 355.6810K for the invocation period of the video task of $20ms$ and when the initial temperature is not more than T_0^∞.

5.6.3 Worst-Case Temperature Analysis under Scheduling Nondeterminism

In this subsection, we we analyze the effect of changes in invocation periods and jitter on the maximal temperature. In addition, we provide hints on how to design a system which is schedulable and meets temperature constraints at the same time. As the maximal temperature depends on task arrivals but not on the scheduling policy, e.g., earliest deadline first (EDF), fixed priority, preemptive or non-preemptive, we restrict ourselves to EDF in the reminder of this section. Note that $\tau = 1.2$s is carefully chosen to make precision loss less than $0.1K$ according to lemma 5.8.

5.6.3.1 Temperature-Aware QoS Optimization

The proposed analytic framework can be used to study the influence of critical design parameters at early design stages. As an example, for the video codec in Table 5.1, a shorter task period provides a higher quality of service (more frames per second). But, in general, shorter periods also lead to higher peak temperature as shown in the results in Figure 5.8. Designers can quickly investigate the effect of such parameter changes by the proposed analysis framework and check if the current system configuration violates temperature constraints. In Figure 5.8, an invocation period of 20ms for instance would imply a peak temperature of 360.18K when the maximum jitter is 60ms. For an invocation period of 40ms, for instance, the video quality will be lower, but also the peak temperature will be as low as 346.09K with the same jitter bound.

5.6.3.2 Changing Task Jitter

Besides varying the invocation period, we vary the maximal jitter in task arrivals. As shown in Figure 5.8, we observe that a large jitter will increase the worst-case temperature. This is an expected qualitative behavior, since a large jitter increases the size of a burst of active modes in the power profile, thus inducing a higher temperature. If such a jitter leads to an unacceptable temperature, designers can redesign the system such at it reduces the jitter by introducing traffic shapers or other resource servers, see e.g., [WMT06].

5.6.4 Schedulability Analysis

In addition, we run an EDF schedulability test by means of the RTC toolbox [WT06b]. It verifies that all output arrival curves satisfy the deadline, i.e., $\sum_i \alpha(\Delta - d_i) \leq \beta(\Delta), \forall\Delta$, with d_i the relative deadline, see e.g., [WT06a].

Figure 5.8 includes the results of such a schedulability test . For instance, for periods of more than 30ms, independent of jitter, the system is

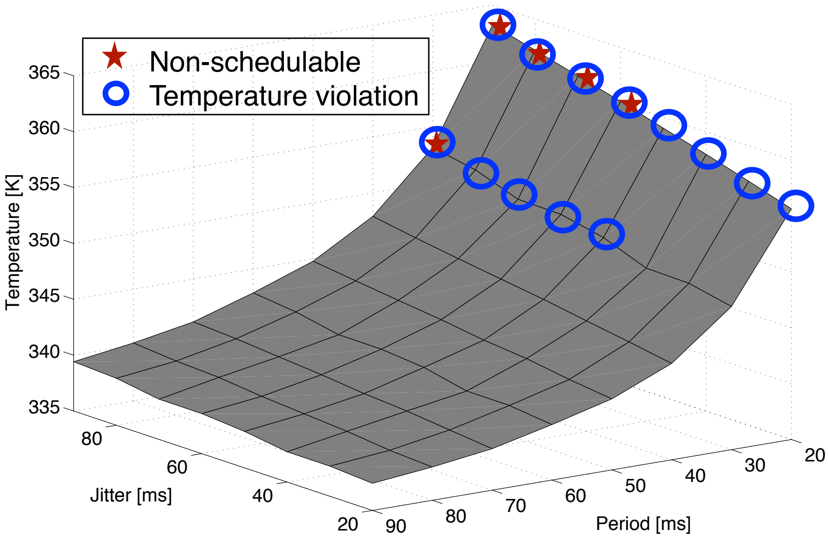


Figure 5.8: Worst-case temperature function of both task invocation period and jitter. Star markers represent non-schedulable sequences, while circle markers highlight the cases that violate the temperature constraint of $350K$.

schedulable. When the video task has the period of 30ms, it is only schedulable when the maximum jitter is less than or equal to 50ms, but only for very small jitter. Note that with a maximum acceptable temperature of 350K, temperature constraints are met for all schedulable systems. All other configurations are not schedulable and they violate the temperature bound of 350K.

For illustration, Figure 5.9 provides an example of a demand bound function for the case where the system is schedulable, i.e., the sum of arrival curves shifted by the corresponding deadlines is smaller that the unit function Δ. Therefore, our analysis can be used as a tool to design systems which are at the same time schedulable and in a temperature-safe region.

5.6.5 General Resource Availability

The proposed technique can be also applied to more general system where the resource may not be always *completely* available for computation. Most modern embedded processors, for instance, support several operation frequencies and deep power down modes for power efficiency. In this subsection, we analyze the effect of different resource availabilities on the worst-case temperature.

Resource availability can be described using *service curves* as explained already in Section 5.4.1. Figure 5.10 shows several service curves as used in the experiments. The full service model ($\beta^u(\Delta) = \beta^l(\Delta) = \Delta$) as well as two lower resource availability curves ($\beta(\Delta) = 0.67$ or $0.33 \cdot \Delta$) are shown on the left-hand side. These service curves may correspond to

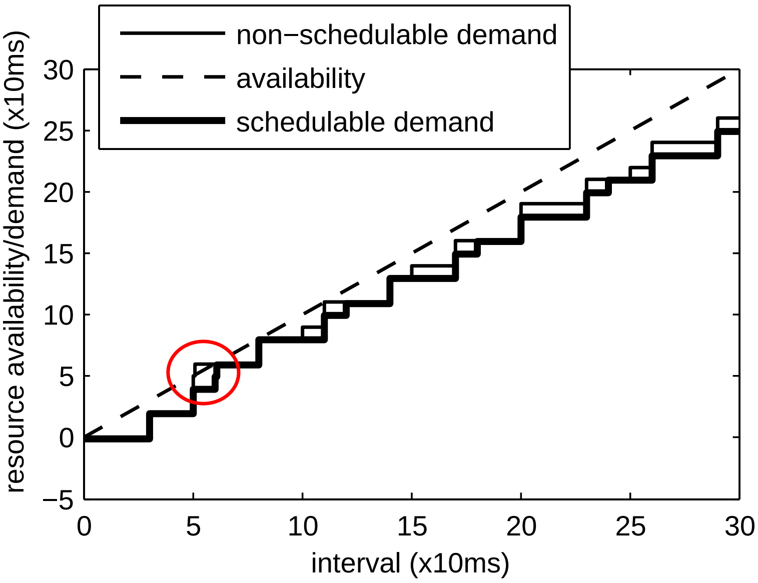


Figure 5.9: Two examples of system traces and schedulability tests. Thin solid line represents a non-schedulable sequence that crosses the availability line at the location indicated by the circle. The thick solid line corresponds to a system which is schedulable.

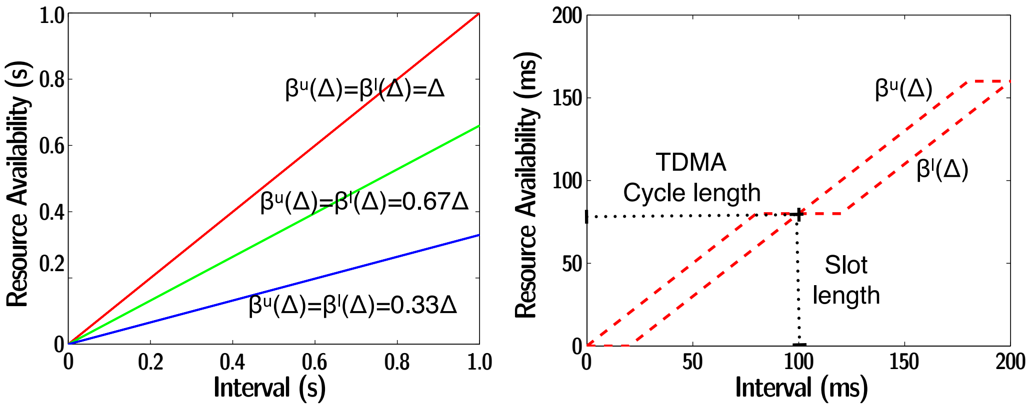


Figure 5.10: Various resource availabilities: Frequency modulated processors (left) and TDMA(right).

operation frequencies of 100%, 67%, and 33%, respectively. They may also be interpreted as the result of a general processor sharing (GPS) scheme where the processing resource is multiplexed between several consumers. The right hand side of Figure 5.10 represents the service curve of a TDMA (time division multiple access) scheduling discipline. A TDMA resource is characterized by two variables: cycle length C and slot length S. For the service curve in the figure, for instance, we chose $C = 100$ms and $S = 80$ms, i.e., the processor is available for consecutive time units of 80ms during every cycle of 100ms.

For this set of computational models, we use the video conferencing example again, but with a different configuration. The period and maximum jitter of the video task are 60ms and 20ms, respectively. The observation time τ is set to $1.2s$. It is worth noting that the same supplied voltage is assumed for all resource availability configurations.

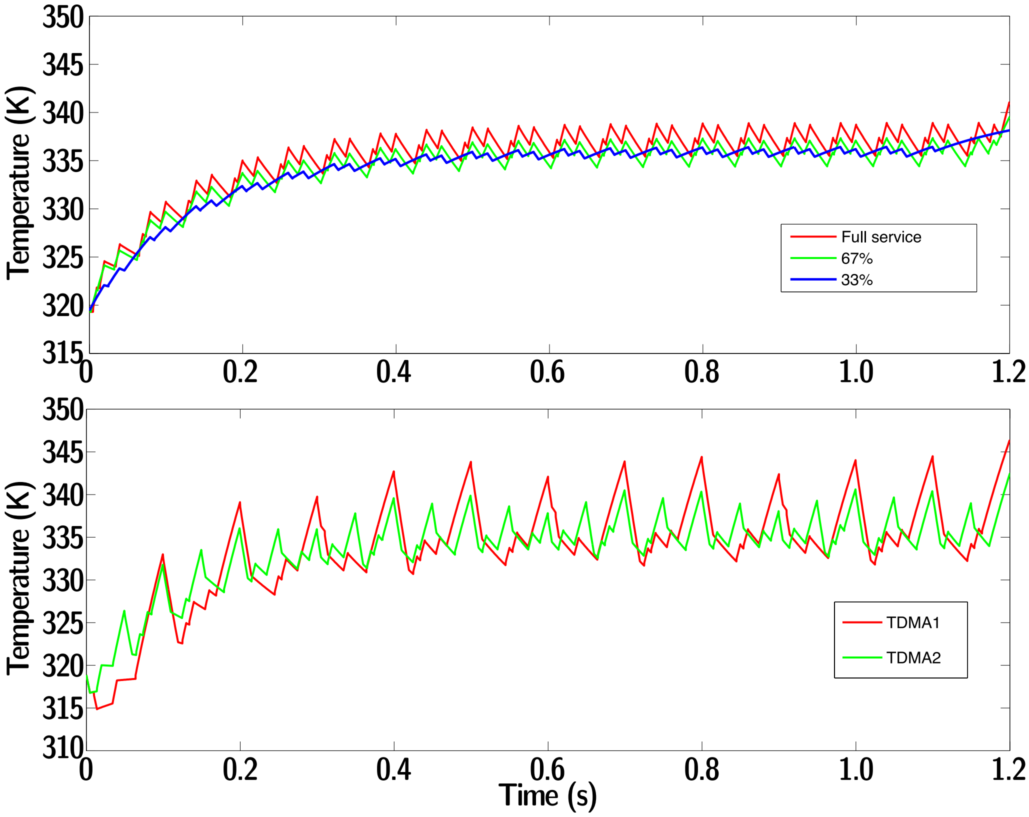


Figure 5.11: Transient temperature changes of the video conferencing example. Period and maximum jitter are set to 60ms and 20ms, respectively.

The first graph of Figure 5.11 provides the comparison of the full service model to the partially utilized ones (67% and 33%). The steady-state temperatures in case of arbitrarily high workload are as follows: $T^{\infty} = 402.37$K for $S = 1$, $T^{\infty} = 367.76$K for $S = 0.67$, and $T^{\infty} = 340.63$K for $S = 0.33$. On the other hand, peak temperatures for the corresponding three service curves are $341.05K$, $339.54K$, and $338.15K$, respectively. The differences in the peak temperatures are not significant compared to that of the steady-state temperatures. When the resource is not sufficient to process the injected workload, the remaining workload is buffered in the waiting queue for further service. This buffering, in turn, causes a pervasive distribution of workload. In contrast, we have some $idle$ intervals to cool down the system in case of a high processing rate. Moreover, the schedulability test fails in case of 33% maximal frequency.

Supposing we must prevent the system temperature from exceeding the critical temperature 340K. In this case, the naive pessimistic analysis drives us to have a dynamic thermal management technique enabled without guaranteeing the real-timeliness. It has also been shown that none of the frequency modulation down to 33% and 67% guarantees the safe peak temperature. On the other hand, the estimates out of the proposed technique confirm that 66% resource availability is a good compromise between the real-time and the peak temperature guarantees. In case of the critical temperature of 350K, no thermal management technique is needed.

The bottom part of Figure 5.11 shows the transient temperature changes when the TDMA scheduling discipline is adopted. We test two configurations: TDMA1 with $C = 100$ms $S = 80$ms and TDMA2 with $C = 50$ms,$S = 40$ms, i.e., half the cycle length but the same utilization. In both cases, only 80% of the processing resource is consumed for processing the workload. Intuitively, the TDMA schemes may lead to a lower peak temperature in comparison to a full resource availability. However, TDMA1 not only shows a higher peak temperature (346.32K) than the full service model but also fails to pass the schedulability test. Again, this is due to the previously mentioned buffering effect. During the idle service interval, the arrived workload is buffered in the queue causing a bursty resource utilization later on. If we reduce the idle service interval as in TDMA2, the peak temperature is reduced to 342.45K and the system becomes schedulable.

In summary, two resource variations are analyzed in terms of peak temperature: reduce the *rate* of the resource and place idle service intervals properly (TDMA). The analysis framework presented in the chapter can be used as a tool to choose proper resource parameters while guaranteeing schedulability and temperature bounds.

5.7 Closing Remarks

The chapter presented an analytical approach to determine the worst case temperature for any given work-conserving real-time system with general resource availability. We considered several power-temperature models that consider temperature dependent power consumption, leakage and thermal conductivity. These lead to non-linear differential equations that describe the relation between power consumption and temperature. The analysis method is able to deal with static and dynamic power consumption and takes into account its temperature dependence. The accumulated workload arriving from all task invocations is characterized by an arrival curve, i.e. by an upper bound on the sum of task execution times arriving in any time interval. Analogously to the workload, the resource availability is modeled by service curves.

The complexity of the analysis comes from the fact that task arrivals and resource availability may be non-deterministic, i.e., due to unknown initial phases, jitter, burst, and provided computation. It is shown that the method proposed in this chapter (a) gives tight upper bounds on the maximal temperature, (b) is constructive in the sense that the worst case arrival of tasks can be determined and (c) provides bounds on the length of the observation interval for a given precision.

Overall, the proposed analysis method can be used during design time to estimate the worst-case peak temperature that may be experienced by the given system. As a result, it may be possible to avoid those design solutions which will cause the processor to overheat and trigger reactive control mechanisms inside the processor (e.g., DVFS) which cause unforeseen loss in performance of the system. If required, the system designer may proceed to investigate the remaining feasible solutions in greater detail using the thermal models obtained in the previous chapters.

One must keep in mind that this chapter uses a higher level of abstraction as compared to previous chapters. As an example, the approach in the previous chapters require discrete time utilization trace(s) both for estimating and constructing the models; whereas this chapter uses abstract interval domain representations of workloads and available computing resources. Thus, given a set of applications and their corresponding schedules, the techniques presented in this chapter require reasonable abstractions from traces to interval domain curves. Additionally, although the current chapter uses simple single pole models in order to keep the analysis tractable, thermal simulations do not necessarily require such simplified models, and models derived from the previous chapters can be used instead.

6

Tolerating Faults in Time Constrained Systems

Summary

This chapter proposes a new technique to transparently tolerate faults in a given multiprocessing system. The emphasis is on designing a fault tolerant real time system which can continue to operate correctly, both in terms of value and timing properties associated with its output(s), even when one (or more) faults have been experienced by the given system. The novelty of the approach lies in detecting and tolerating faults, *without* using any timer resources. Furthermore, the proposed approach *only* requires the knowledge of interface level timing properties of applications, making it applicable to complex, and even legacy applications. Therefore, the technique can also be applied to safety critical applications which have already been certified, but are not fault tolerant; or those applications wherein any modifications of the software is not feasible.

6.1 Introduction

Even after adequate design measures have been taken, a system may still suffer from a fault, leading to a failure, either because of one or more design assumptions have been violated (e.g., the load on the processor exceeds the bounds assumed for analysis, see Chapter 5), or the model(s) (e.g., thermal model) on which the design analysis was based may be too erroneous. Furthermore, discussions in the previous chapters focused on the problem of avoiding thermal faults, whereas in practical use cases, a system may experience a fault due to a myriad of other factors, such as cosmic particles, hardware errors, or even software bugs.

Accordingly, this chapter presents a new fault tolerance technique specifically targeted to time constrained systems (e.g., real time systems), enabling such systems to tolerate faults while simultaneously meeting all performance requirements. Specifically, using the technique proposed in this chapter, a system can tolerate faults transparently to the observer, while continuing to compute outputs which are correct in value, and also conform to the expected timing properties, such as throughput or deadlines.

We assume that a given system (e.g., a function, an application, or even a complete network) to be made fault tolerant fails *silently*. In other words, we assume that the given vulnerable system possesses the following properties: (*i*) in the fault-free scenario, the system computes the correct output, both in value and timing properties, which can be (optionally) verified by independent observers, and (*ii*) in the case of a fault, it ceases to produce any outputs, see [HSS96].

The assumption of the fail silent system property is not unusual. In fact, modern safety critical systems are often designed to be fail silent. Common examples are the FlexCAN, and the Time Triggered Protocol Class C (TTP/C) communication architectures, see [Ber06, KB03]. Therefore, a number of techniques already exist, both at the application level and at the hardware level, which ensure that all faults are exhibited solely as timing faults. Brasileiro *et. al.* describe the construction of a fail-silent system at the application level, whereas a patent provides an example of how processors are now designed to enforce fail-silent behavior, see [HSL78, Mil98] for references.

Since we assume a fail silent system, the overall problem of fault detection and tolerance reduces to detecting and tolerating timing faults. Specifically, a system (or a part of it) exhibits a timing fault when one or more of its inputs or outputs fail to meet the desired timing properties, such as rates, or deadlines. However, *efficiently* detecting and tolerating timing fault remains a major challenge.

In cases where applications exhibit simple timing behavior, (e.g., strictly periodic applications), timeout (e.g., watchdog) based solutions may be used. Such simple approaches are not effective for applications based on general dataflow process networks, which are usually asynchronous and can have bursty timing characteristics. Such process networks are generally used to design and implement streaming applications (e.g., multimedia). Detecting timing faults is particularly difficult in such process networks, since data packets (or tokens) produced by the given fault free-system may show some legitimate burstiness (i.e., more than normal amount of data tokens are received in a given time window) or jitter (i.e., a data token either arrives earlier or later than expected).

A few approaches for detecting timing faults under such circumstances have been proposed. One approach is to use a set of timers for monitoring one or more timing properties of each input or output stream produced by the given vulnerable system, e.g., inter-arrival times of between consecutive data tokens, or throughput, see [HCBK12]. In principle, this approach can be applied for detecting timing faults in systems with complex timing behavior. However, such an approach does not scale with the complexity of the timing patterns to be monitored for a fault, in terms of the number of timer resources and monitoring effort required. Another approach is to use *l-distance* functions in which the time-separation between $(l+1)$ consecutive events is monitored. Reduction in the number of timer, computational, and memory resources required is accomplished by approximating timing curves (e.g., arrival curves) into a more restrictive *minimum distance function*, which also carries with it a possibility of false positives, see [NMA$^+$12]. Another solution is to design the system in which processes share their internal states with each other, and fault detection strategy is based on continuously monitoring the internal states of one or more processes, see [GDJ12]. A major drawback of this approach is the additional requirement that a system be specifically designed (or modified) such that its internal states are observable. Additionally, there may be heavy computational burden involved in concluding the fault status of the application based on observing several states. Therefore, it is not clear how the approach scales with the number of processes in the application.

With a focus on process networks with timing constraints, this chapter takes a new approach to detecting *and* tolerating timing faults. We treat the given vulnerable system as a black-box, whose interface-level timing models are either available, or can be generated quickly from calibrations. This makes the proposed approach applicable to large and complex systems, and also does not impose any special design requirements (such as observability of internal states) to be met by a system in order to be compatible with the proposed approach.

Succinctly, we assume that we have two (or more) functionally equivalent versions (henceforth, *replicas*[1]) of the vulnerable system, with sufficient design diversity to avoid common model faults. These replicas execute in parallel, operating on the data provided by a common set of input stream(s), with their respective outputs being *merged* (details follow) into final output stream(s).

In contrast to the conventional approach of monitoring the timing properties of input and output streams, we monitor the number of data tokens in the input and output FIFO buffers of each replica. Using the available timing models, we can compute the expected bounds on the number of tokens in each of these input and output FIFO buffers, under the assumption that both replicas are fault-free. Subsequently, assuming only a

---

[1] The term is slightly abused here. For this chapter, two or more replicas only indicate functional and timing equivalence between them. However, each replica may have a different implementation, and therefore, replicas are not necessarily bit-identical.

single permanent timing fault, we compare the number of tokens in corresponding FIFO buffers at the inputs as well as the outputs of the replicas. Since the replicas are functionally equivalent, and must also satisfy a common set of timing properties at their respective interfaces, the difference in between number of tokens in the corresponding FIFO buffers of each replica must remain within pre-computed bounds. Therefore, the difficult problems of detecting a timing fault can be reduced to simply deciding whether the difference in the number of tokes in the corresponding FIFO buffers exceeds a pre-computed threshold.

For further discussion, we first define the problem considered in this chapter:

Provide a provably correct and efficient mechanism for detecting and tolerating single timing faults in real-time process networks.

As hinted earlier, we use *active replication* for tolerating timing faults, and for simplicity, we focus on tolerating at most one permanent timing fault, using only two replicas of a given real time data flow process network. This restriction can be easily relaxed by adding more replicas to the system. The main contributions of this chapter are summarized as follows:

1. Design and analysis of arbitration mechanisms for a duplicated real-time process network such that single timing faults can be efficiently tolerated, and
2. Memory and time efficient fault detection algorithms developed from timing models which do not require any runtime timekeeping or significant computations.

The proposed approach differs from the *l*-distance based technique in two aspects: (*i*) no use of any timer resources, and (*ii*) no false positives since the approach does not make any over-approximations (such as the restrictive distance function technique).

6.2 Motivational Example

It has already been discussed that detecting timing faults by present methods is hard. We now show that even transparently tolerating timing faults is also not trivial. Consider a simple process network shown on the top side of Figure 6.1 that contains processes and communication channels with FIFO semantics.

A portion of the process network, called the *critical subnetwork* is duplicated (i.e., two replicas are created) for fault-tolerance. The set of producer process(es), P, provide data tokens to the critical subnetwork(s), and the consumer process(es), C, consume tokens from this subnetwork. A *replicator channel* duplicates the same stream from a producer to each replica, whereas a *selector channel* combines the streams from the replicas into a single input stream for a consumer. In the chapter, we refer to the process network with un-replicated critical subnetwork as *reference*, and to the process network with two replicas of its critical subnetwork as *duplicated*.

Transparent fault tolerance requires that both the reference and duplicated process networks behave equivalently when observed at their respective input and output interfaces, even when one of the replicas suffers from a single permanent timing fault. We assume that the sequence of data tokens produced by a process and the process network is independent of the timing of the network (e.g., following the Kahn Process Network semantics). All FIFO buffers have bounded capacities, and processes have blocking semantics. Therefore, a process attempting to write tokens to a full output FIFO buffer, or attempting to read tokens from an empty input FIFO buffer will block, until the said operation can be successfully completed. For simplicity, assume that only the critical subnetwork may suffer from a permanent timing fault.

Arbitration (or Merging) Streams at a Selector Channel

With only two replicas, arbitration by majority voting is ruled out. An option is to have a dedicated fault detection mechanism which informs the selector of any fault in either replica, allowing the selector to block all outputs from the faulty replica from reaching the consumer. However, this option places additional constraints on the system, requiring a dedicated and reliable monitor(s). It is therefore desirable that the selector can autonomously and *efficiently* detect timing faults in the replicas. Notice that asynchronous and bursty characteristics of the process network makes it very difficult to apply naive timeout (e.g., watchdog) based approaches to fault detection.

Deadlocked Non-Faulty Replicas

Assume that the selector has detected a timing fault in the top replica by some mechanism. Suppose that as a result, the selector stops destructively reading tokens from this subnetwork, which in turn may stop reading tokens from its input, causing the FIFO buffer at the output replicator to fill up, eventually causing one or more producers to block. This in turn starves the lower (correctly working) subnetwork from processing further tokens, causing the selector to flag this subnetwork also as faulty, compromising the reliability of the entire system. A simple solution to this problem is to have the selector (or some monitor) inform the replicator about the fault status of the replicas within a bounded time, but then, as pointed out earlier, this imposes additional constraints on the system. Alternatively, the replicator could allow for a non-blocking write by the producer process P, but this solution requires the replicator channel to potentially be able to store an unbounded number of tokens.

6.3 Notations and Model

For simplicity, we consider a simple dataflow process network with one critical subnetwork connected to a single producer and a single consumer via a FIFO buffer on either side, see Figure 6.1. All presented discussion and analysis is equally applicable to a general model with the critical subnetwork having multiple input and output channels. The input and output ports of the critical subnetwork are denoted by $\mathcal{I}$ and $\mathcal{O}$, respectively. Communication between processes is done via read and write operations on FIFO channels with finite capacities; and the processes have blocking semantics. The capacity of a FIFO buffer F_i is denoted by $|F_i|$. We require that a timing fault does not lead to wrong data (value of a token) in the application, and hence we assume that the process network is *determinate*, i.e., the sequence of tokens and their values produced by a process network is dependent only upon the sequence of input tokens, and not upon the timing of token availability. The producer, the consumer, and the reference and duplicated process networks have associated real time properties. Their timing properties can be, for example, specified in terms of *arrival curves* or any other real-time model. Details on arrival curves are presented in Section 6.4, and can also be found in [CLS$^+$06].

The fault tolerant system is constructed by duplicating the critical subnetwork, into replicas R_1 and R_2, along with necessary FIFO buffers and channels. A special *replicator* channel duplicates the stream from the producer process to the corresponding input ports of the replicas, $\mathcal{I}_1$ and $\mathcal{I}_2$, respectively. Similarly, a *selector* channel arbitrates (merges) the data streams from the output ports of the replicas, i.e., $\mathcal{O}_1$, and $\mathcal{O}_2$ and provides the resulting stream to a system output process C. A token produced by a replica R_k on its output channel is denoted as $T_k[j]$, where $j \in \mathbb{N}^+$ is the monotonically increasing *sequence number* of the said token. A function $t : \mathbb{N}^+ \times \mathbb{N}^+ \to \mathbb{R}_{\geq 0}$ provides the timestamp of a token $T_k[j]$, given as $t(k, j)$, indicating the time instant when the token was produced.

We assume that owing to hardware costs, only a part of the system can be made reliable, see [TGC$^+$06]. Thus, only processes and channels within replicas R_1 and R_2 are considered to be unreliable, whereas the rest of the system is assumed to execute on reliable hardware. We also assume that the system can experience at most a single timing fault, which is eventually observed when the faulty replica either stops producing (or consuming) tokens, or does so at a rate lower than expected.

6.4 Proposed Solution

Section 6.4.1 proposes the design of the replicator and the selector processes to be used for constructing a duplicated process network equivalent to the reference process network, both in functionality and timing (Section 6.4.2). Section 6.4.3 discusses fault-detection mechanisms in the replicator

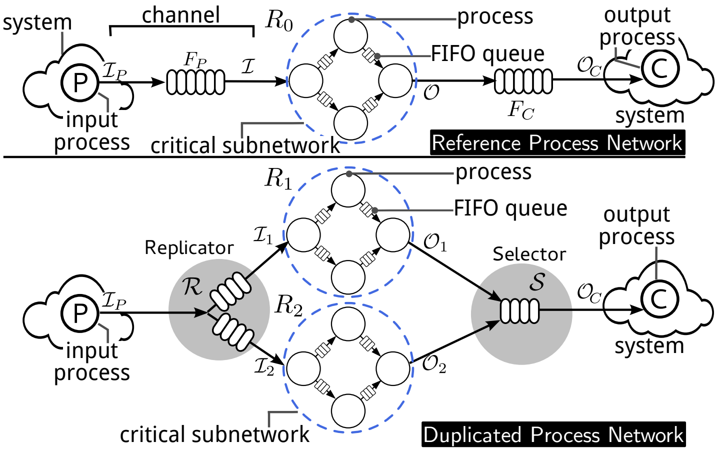


Figure 6.1: The reference and duplicated process networks. For simplicity, the critical subnetwork has only one input and output channel(s).

and the selector, also extending the discussion to a bounded memory system. Finally, Section 6.4.4 presents necessary mathematical details.

6.4.1 Replicator and Selector

We assume all read and write operations to the replicator and the selector channels to be atomic. A replicator channel $\mathcal{R}$ has two reading interfaces and a single writing interface, and is described by the following rules:

1. It contains two FIFO buffers of sizes $|\mathcal{R}_1|$ and $|\mathcal{R}_2|$ respectively, one for each reading interface. Each buffer has a space and fill variable which are initially set to $\mathit{fill}_1 = \mathit{fill}_2 = 0$ and $\mathit{space}_1 = |\mathcal{R}_1|$, $\mathit{space}_2 = |\mathcal{R}_2|$.

2. Each reading interface of the replicator has a destructive and blocking read access to the corresponding buffer. A read event increments the corresponding space variable and decrements the corresponding fill variable.

3. If $\min\{\mathit{space}_1, \mathit{space}_2\} > 0$, then a *write* event to the write interface buffers a token to both FIFO buffers, decrements space_1 and space_2, and increments fill_1 and fill_2, else the write to the replicator is blocked.

In other words, the replicator channel duplicates every input token to both FIFO buffers, each one linked to a read interface. More efficient memory management schemes are possible, but we retain the simple design for the present discussion.

A selector channel $\mathcal{S}$ has two writing interfaces and a single reading interface and is described by the following rules:

1. There are two space variables $space_1, space_2$ and a single variable $fill$ associated with the buffer. Initially, we have $fill = 0$ and $space_1 = |\mathcal{S}_1|, space_2 = |\mathcal{S}_2|$. The selector maintains only a single FIFO buffer for the channel, with size given by $|\mathcal{S}| = \max\{|\mathcal{S}_1|, |\mathcal{S}_2|\}$.
2. The reading interface of the selector has a destructive and blocking read access to the buffer. A $read$ event increments all space variables and decrements the fill variable.
3. A $write$ event to a write interface such as interface 1 blocks if $space_1 = 0$. Suppose now that $space_1 > 0$. If $space_1 \leq space_2$, then the token to be written by interface 1 is buffered in the FIFO, $fill$ is incremented and $space_1$ is decremented. If $space_1 > space_2$ with $space_2 = 0$, then just $space_1$ is decremented, leaving $space_2$ unmodified. Furthermore, the corresponding late arriving token to write interface 2 is dropped.

In other words, the selector contains two virtual buffers, one for each writing interface. Under fault-free conditions, both replicas provide the same sequence of tokens to the selector, and selector must buffer a token from a write interface which provides the first token of each duplicate pair. Therefore, the selector buffers a token from interface 1 if $space_1 \leq space_2$, else it buffers from interface 2. A process can successfully read from the selector FIFO if $fill > 0$. It must be noted that under no-fault conditions, the FIFO buffers are sized such that the following always holds: $space_1 > 0$ and $space_2 > 0$. The details of achieving the required buffer dimensions follow in this chapter.

6.4.2 Equivalence

We show that if the FIFO buffers in the replicator are unbounded, the duplicated process network is equivalent to the reference process network, both in functionality and timing, even if one replica suffers a single timing fault. First, we present necessary definitions and a lemma.

A sequence of tokens produced by replica R_k is denoted as $\mathcal{Q}_k = \langle T_k[1], T_k[2], T_k[3], \cdots \rangle$. Prefix ordering between sequences is denoted as $\mathcal{Q}'_k \sqsubseteq \mathcal{Q}_k$. The sequence of timestamps associated with $\mathcal{Q}_k$ is given by $t(\mathcal{Q}_k) = \langle t(k,1), t(k,2), t(k,3), \cdots)\rangle$.

The consumer process expects the input stream(s) to satisfy one or more timing constraints at its input interface, such as token inter-arrival timings. In practice, this imposes certain conditions on timestamps associated with each data token in the input stream to the consumer. Assume that all sequences of timestamps associated with $\mathcal{Q}_k$ satisfying the timing requirements of the consumer are represented by the set $\mathcal{T}_C$ of sequence of timestamps, i.e., $t(\mathcal{Q}_k) \in \mathcal{T}_C$. We assume that if a sequence $\mathcal{Q}_k$ with timestamps $t_1(\mathcal{Q}_k)$ satisfies the requirements of the consumer, $t_1(\mathcal{Q}_k) \in \mathcal{T}_C$, then the same sequence with different timestamps $t_2(\mathcal{Q}_k)$ also satisfies the requirements of the consumer if some of the tokens arrive *earlier*, i.e., $t_2(\mathcal{Q}_k) \in \mathcal{T}_C$, where:

$$\forall t_1(k,j) \in t_1(\mathcal{Q}_k), \forall t_2(k,j) \in t_2(\mathcal{Q}_k) \mid t_2(k,j) \leq t_1(k,j) \tag{6.1}$$

We also assume that replicas must satisfy the timing characteristics of the consumer. Therefore, in a duplicated process network, timestamps $t(\mathcal{Q}_1)$ and $t(\mathcal{Q}_2)$ produced by replicas R_1 and R_2, respectively, must satisfy $t(\mathcal{Q}_1) \in \mathcal{T}_C$ and $t(\mathcal{Q}_2) \in \mathcal{T}_C$.

A pre-requisite to equivalence between the duplicated and the reference process networks is that the selector *isolates* the replicas from each other:

Lem. 6.1. *The selector prevents the output of one replica from affecting the output of the other, both in value and time.*

Proof. From the properties of the process network, and those of the selector, a replica, say R_2 can *only delay* the tokens from the replica R_1. This delay would be due to any back-pressure caused by R_2, which is experienced by R_1. However, from rule 3 of the selector, the only variable that governs the back-pressure felt by R_1 is $space_1$. From the construction of the selector, the $space_1$ variable is never modified by write interface 2 (and vice versa), and hence, the back-pressure felt by R_1 is never caused (or contributed to) by R_2. The lemma follows. □

The functional and timing equivalence between the duplicated and reference process network is shown next:

Thm. 6.1. *If the replicator has unbounded FIFO buffers, then a sequence $\mathcal{Q}_P$ with timestamps $t(\mathcal{Q}_P)$ provided to the reference and duplicated process networks results in the same output sequence $\mathcal{Q}_C$ from both the reference and duplicated networks, even if the duplicated process network suffers a single timing fault. Furthermore, if the timestamps of the sequence generated by the reference process network $t(\mathcal{Q}_C) \in \mathcal{T}_C$, then the sequence of timestamps $t'(\mathcal{Q}_C)$ generated by the duplicated process network is also in $\mathcal{T}_C$.*

Proof. Since the replicator FIFO buffers are unbounded, $\min\{space_1, space_2\} > 0$ (rule 3 of the replicator) is always true, and consequently, a replicator channel always duplicates each token to both input ports $\mathcal{I}_1$ and $\mathcal{I}_2$ of the replicas. Furthermore, the replicator does not change the timestamp of a token when it inserts it into both FIFO buffers. Thus, a sequence $\mathcal{Q}_P$ with timestamps $t(\mathcal{Q}_P)$ at the write interface of the replica always results in the same sequence $\mathcal{Q}_P$, with the same timestamps, at $\mathcal{I}_1$ and $\mathcal{I}_2$.

Under no fault conditions, the replicas are determinate but nondeterministic in timing characteristics, therefore, given the same input sequence $\mathcal{Q}_P$ with timestamps $t(\mathcal{Q}_P)$, the replicas produce at their output ports output sequences $\mathcal{Q}_1 = \mathcal{Q}_2$, with non-equal sequences of timestamps $t(\mathcal{Q}_1) \neq t(\mathcal{Q}_2)$ respectively.

Next, the selector evaluates which replica has provided the most recent token of a duplicate pair, by evaluating $space_1 \leq space_2$. If $space_1 \leq space_2$, then the replica R_1 has provided the first token of the most recent duplicate pair, which is buffered into the FIFO, and the selector simply discards the corresponding late arriving token from R_2. In other words, the selector buffers the earlier arriving token from each duplicate pair into its FIFO, resulting in a sequence $\mathcal{Q}_C = \mathcal{Q}_1 = \mathcal{Q}_2$ with timestamps $t(\mathcal{Q}_C)$. Since $t(\mathcal{Q}_1) \in \mathcal{T}_C$ and $t(\mathcal{Q}_2) \in \mathcal{T}_C$, then as in (6.1), we have $t(\mathcal{Q}_C) \in \mathcal{T}_C$ (also see Lemma 6.1).

If a replica R_1 experiences a timing fault at any instant t, then eventually, we have $\mathcal{Q}_1 \sqsubseteq \mathcal{Q}_2$ and $space_2 \leq space_1$, and the selector simply buffers the tokens from the replica R_2. The timestamp of a token missing in $\mathcal{Q}_1$ but with a corresponding token in $\mathcal{Q}_2$ is taken to be infinity, and therefore timestamps of the tokens produced by the selector subsequent to a fault correspond to those from R_2.

For comparison, given $\mathcal{Q}_P$, the reference process network produces a sequence $\mathcal{Q}_C = \mathcal{Q}_2$, which is the same output as the non-faulty replica R_2 produces (since the replicas are determinate and are derived from the reference process network). Furthermore, *if* the reference process network meets the timing requirements of the consumer, then $t(\mathcal{Q}_C) \in \mathcal{T}_C$. □

6.4.3 Fault Tolerance with Bounded Memory

Thus far, we discussed the operation of the selector and the replictor channels without considering any limits of the FIFO buffers associated with these entities. In order to be practically useful, we will now describe fault tolerance with bounded memory. We assume that the reference process network has been designed correctly, i.e., all FIFO buffers have been sized appropriately such that a producer does not block on a full FIFO buffer, and a consumer does not stall on an empty FIFO buffer. Under this assumption, this section describes fault detection assuming bounded FIFO buffers inside the replicator and the selector channels. We follow up with the approach that can be used to compute the minimum FIFO buffer size required for the selector and the replicator channels.

Since the replicator channel is transparent to both replicas, and the selector channel decouples replicas from their output sides, it follows that if in the reference process network R_0 requires an input FIFO buffer of capacity $|F_P|$, then the FIFO buffer $\mathcal{R}_1$ in the replicator channel requires a capacity $|\mathcal{R}_1| = |F_P|$ for $R_1 = R_0$ (and similarly for the selector FIFO buffer), see Figure 6.1. Furthermore, if in the reference process network the producer does not block on full FIFO buffer(s), the same producer will also not block on a full replicator FIFO buffer, as long as the replicas are not faulty (and similarly for the consumer). Therefore, in order to ensure that the reference process network and the duplicated process network are equivalent even when the latter experiences a single timing fault, it is required that in the duplicated process network, the producer never blocks on a full replicator FIFO buffer associated with the faulty replica (the selector channel already has bounded memory). It follows that functional and timing equivalence between duplicated and reference process network requires that the replicator channel be able to autonomously detect a timing fault. Once the replicator detects a timing fault, it can "shut down" the faulty replica simply by stopping any further writes to the corresponding write interface. This prevents the replicator from blocking at any of its write interface(s), and in turn prevents the producer process from blocking at its output FIFO buffer.

Though fault detection at the selector channel is not strictly necessary, we nevertheless describe a way to autonomously detect a timing fault at the selector channel. The capability of the selector and the replicator channels to autonomously detect a timing fault can be leveraged to quickly spread the information about the fault status of the system using multiple sources.

Fault Detection at the Replicator Channel

First note that the replicator FIFO buffers with capacities $|\mathcal{R}_1|$ and $|\mathcal{R}_2|$ should never overflow under fault free conditions. Therefore, a replica, say R_1 is deemed faulty if the actual number of tokens in the associated FIFO exceeds $|\mathcal{R}_1|$, causing the producer to block on the full FIFO. In other words, if $space_1 = 0$ when the producer attempts to write a token, then the replica R_1 is faulty. Similar arguments also apply to the case with replica R_2. We introduce variables $fault_1$ and $fault_2$ for the replicator channels, each initialized to `FALSE`. If $space_1 = 0$ when the producer writes a new token to the replicator, then $fault_1 =$ `TRUE`, and the replicator does not insert new tokens into this FIFO (and similarly for $fault_2$ and $space_2$)

Fault Detection at the Selector Channel

There are two methods for detecting a fault at the selector. The first method is simple: the replica R_1 may stall the consumer (and is hence faulty) if $space_1 > |\mathcal{S}_1|$, and similarly for R_2. The second approach is based on the intuition that if both replicas serve and satisfy timing bounds imposed by a common consumer, then the outputs from both replicas must not diverge too much from each other. The divergence is quantified by the difference in total number of tokens received by the selector over both input channels. Therefore, the selector monitors the difference $|space_1 - space_2|$ and if the difference exceeds a threshold D, then the replica R_1 is faulty if $space_1 > space_2$, else R_2 is faulty. The details will be elaborated in the next section. The rule 3 of the selector can be easily modified to include fault detection at the selector channel.

6.4.4 FIFO Conditions and Threshold Calculations

FIFO Capacities and Initial Fill Conditions

Let $\mathcal{G}_P[s,t)$ denote the total number of tokens generated by a producer in the interval $[s,t)$. Then, the upper and lower *arrival curves*, $[\alpha_P^u, \alpha_P^l]$ denote the maximum and minimum number of tokens generated by the producer in *any* time interval $\triangle$, see [CLS$^+$06]:

$$\alpha_P^l(t-s) \leq \mathcal{G}_P[s,t) \leq \alpha_P^u(t-s) \;\; \forall s < t \tag{6.2}$$

Equation (6.2) is either provided as a part of the timing model, or is derived from calibration experiments. Let $[\alpha_{i,in}^u, \alpha_{i,in}^l]$ be the maximum and minimum number of tokens consumed by a replica $R_i \mid i \in \{1,2\}$ in any time interval $\triangle$. We require that the producer never blocks on its output FIFO, i.e., F_P in reference network, and equivalently, FIFO buffers $\mathcal{R}_1$ and $\mathcal{R}_2$ in the replicator channel. The required size of the FIFO $|F_P|$ (equivalently, the capacities $|\mathcal{R}_1|$ and $|\mathcal{R}_2|$) which ensures this is:

$$\alpha_P^u(\triangle) \quad \leq \quad \alpha_{i,in}^l(\triangle) + |F_P| \;\; \forall \triangle \geq 0 \tag{6.3}$$

Notice that it is *acceptable* that the replica(s) may stall on empty FIFO buffers $\mathcal{R}_1$ and $\mathcal{R}_2$ as long as the consumer does not stall on *its* empty input FIFO buffer. That the consumer does not stall on its empty FIFO buffer, i.e., F_C in the reference network, and $\mathcal{S}_1$, $\mathcal{S}_2$ in the duplicated process network, requires an initial number of tokens, $F_{C,0}$:

$$\alpha_{i,out}^l(\triangle) \quad \geq \quad \alpha_C^u(\triangle) - F_{C,0} \;\; \forall \triangle \geq 0 \tag{6.4}$$

where $\alpha_{i,out}^l(\triangle)$ is the minimum number of tokens produced by the replica R_i in any time interval $\triangle$, and $\alpha_C^u(\triangle)$ denotes the maximum number of tokens consumed by the consumer process in any time interval $\triangle$.

Threshold Calculations

We present the calculations only for the selector channel. The corresponding computations for the replicator channel are analogous. We compute the maximum difference in the total number of tokens received from both replicas, D over *any* time interval $\triangle$. The threshold, D, which is then used to indicate a faulty replica, is the smallest integer satisfying:

$$D \quad > \quad \sup_{\forall i,j,\lambda \geq 0} \{\alpha_{i,out}^u(\lambda) - \alpha_{j,out}^l(\lambda)\} \tag{6.5}$$

where sup is the supremum of a set. The equation can be easily verified by applying the definition of arrival curves. Notice that (6.5) guarantees that there are no false-positives.

Fault Detection Times

A replica is considered to have suffered a timing fault when it fails to meet the expected timing properties *at its interfaces*, and *not* when a particular node(s) inside the replica may have experienced a fault. Suppose that at time s, R_1 and R_2 have produced a total of T and $T-(D-1)$ tokens respectively, when R_1 suffers a timing fault. Subsequently, R_2 must produce a $(D-1)+D=2D-1$ tokens more than R_1 before the selector can detect a fault. Let the fault be detected at time t. For maximum fault detection time, let the replica R_2 supply tokens at the lowest possible rate, i.e., its arrival curve *subsequent* to the fault is α_2^l. Let $\overline{\alpha}_1^u$ indicate the upper arrival curve of R_1 *subsequent* to the fault, which still fails to meet the required real time constraints. The maximum time $\triangle$ to detect the fault satisfies:

$$\inf\{\triangle \mid (\alpha_2^l - \overline{\alpha}_1^u)(\triangle) \geq (2D-1)\} \tag{6.6}$$

where `inf` is the infimum of a set. Generalizing, the maximum fault detection time is:

$$\max_{\forall i,j,i\neq j}\{\inf\{\triangle \mid (\alpha_i^l - \overline{\alpha}_j^u)(\triangle) \geq (2D-1)\}\} \tag{6.7}$$

For the case when the faulty replica stops producing any tokens altogether, (6.7) can be simplified to:

$$\max_{\forall i}\{\inf\{\triangle \mid (\alpha_i^l)(\triangle) \geq (2D-1)\}\} \tag{6.8}$$

6.5 Tolerating n Simultaneous Timing Faults

In order to tolerate upto n timing faults simultaneously, we require at least $n+1$ fail-silent replicas of the given application. In the further discussion, we assume $n' \geq n+1$ replicas. The replicator and the selector components will change only slightly in order to accommodate the new fault-tolerant requirement.

The Replicator Channel

The replicator channel has 1 writing interface, n' reading interfaces, n' FIFO buffers of capacities $|\mathcal{R}_1|\ldots|\mathcal{R}_{n'}|$. The associated space and fill variables for read interface i are set as: $space_i = \mathcal{R}_i$ and $fill_i = |\mathcal{R}_i|$. If $\min_{\forall i}\{space_i\} > 0$, then the write event buffers a new token into all FIFO buffers, decrementing the associated $space$ and $fill$ variables appropriately. However, as before, if $\exists i : \min\{space_i\} \leq 0$, the write to the replicator is blocked.

The Selector Channel

The modifications to the design of the selector channel are also similar. The selector channel has n' writing interfaces, 1 reading interface and a single FIFO buffer. There are n' space variables: $space_i \dots space_{n'}$, and a single $fill$ variable. Initially, the variables are set to $fill = 0$ and $space_i = |\mathcal{S}_i|$. The actual capacity of the FIFO channel in the selector is $\max\limits_{\forall i}\{space_i\}$. As before, the reading interface of the selector has a destructive blocking access to the buffer, and the *read* event increments all $space$ variables and decrements the $fill$ variable. The *write* event to an interface i blocks if $space_i = 0$, and as before, the selector inserts the first arriving token from n' duplicates into its FIFO buffer.

Equivalence

Similar to the arguments presented in Section 6.4.2, it is straightforward to see that the new process network constructed out of n' replicas is equivalent to the reference process network, in terms of timing and functionality.

Fault Tolerance with Bounded Memory

Similar to Section 6.4.3, we assume that the reference process network has been designed correctly. The capacity of the FIFO buffer $\mathcal{R}_i$ (from which tokens are read by the replica R_i) in the replicator channel is $|\mathcal{R}_i| = |F_P|$. Here F_P is computed from the reference process network by setting $R_0 = R_i$. The computations for $|\mathcal{S}_i|$ are also similar.

Furthermore, calculations in (6.3), (6.4) and (6.5) are performed separately for each replica, and are therefore independent of the number of replicas in the fault-tolerant process network.

Similar to the duplicated process network, the replicator channel considers that the replica R_i has suffered a timing fault if $space_i = 0$. Observing from the selector channel, a replica is considered faulty if it can stall the consumer. Therefore, if the selector detects that for a replica R_i, the associated $space$ variable satisfies $space_i > |\mathcal{S}_1|$, then the replica R_i is considered faulty. It is also possible to monitor the difference in the total number of tokens received from all replicas in order to determine the fault status of replica R_i. Therefore, the selector evaluates the condition $\max\limits_{\forall j \neq i}\{space_j - space_i\} > D$, where D is the threshold computed from (6.5). If the condition evaluates to true, the replica R_i is declared faulty.

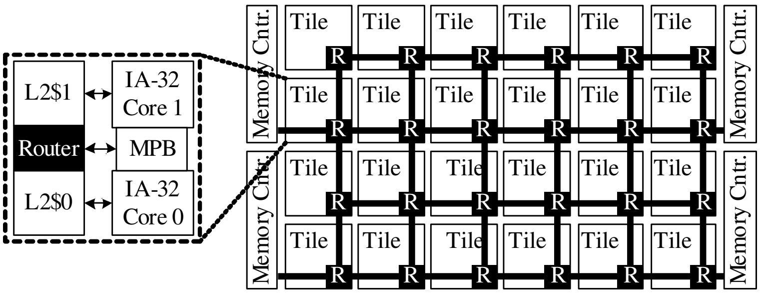


Figure 6.2: Schematic representation of Intel's SCC processor [H$^+$10a].

6.6 Experiments and Results

Hardware Platform

We used Intel's 48-core Single Chip Cloud Computer (SCC) for experiments. The SCC is a prototype of future embedded on-chip many-core platforms [H$^+$10a]. The processor consists of 24 tiles that are organized into a 4×6 grid and linked by a 2D mesh on-chip network. A tile contains a pair of P54C processor cores, a router, and a $16\,\mathrm{KB}$ block of SRAM. Each core runs at $533\,\mathrm{MHz}$ and each router runs at $800\,\mathrm{MHz}$. The on-tile SRAM block is also called "message passing buffer" (MPB) as it enables the exchange of information between cores in the form of messages. Figure 6.2 schematically outlines the SCC processor.

The SCC was configured with the following parameters: tile clock speed: 533MHz, Router clock speed (all routers): 800MHz, DDR3 Memory clock speed: 800MHz, see [H$^+$10b]. The clocks on all cores were synchronized at bootup for consistent timing observations. Real time performance was achieved by executing applications on the baremetal (i.e., no operating system on the cores) SCC, switching off all L2-caches, disabling interrupts, and mapping one process *per tile* to reduce cross traffic at the routers, see [Z$^+$12, ZM12]. The iRCCE non-blocking communication library was used, and all data was sent/received in chunk sizes not exceeding 3KB, ensuring that all messages are routed exclusively via the message passing buffers, see [RSS$^+$13, CLRB11]. The fast on-chip communication does not significantly influence FIFO sizes or fault detection timings.

Applications

Three representative real time process network based applications were used for experiments: (i) a Motion JPEG (MJPEG) decoder, (ii) an adaptive differential pulse code modulation (ADPCM) application (encoder+decoder), and (iii) an H.264 encoder, see Figure 6.3 for details. The design diversity in the replicas is represented by different jitter values, see Figure 6.4. All timing parameters are reported as <period, jitter, minimum inter-arrival distance> tuple, also known as the

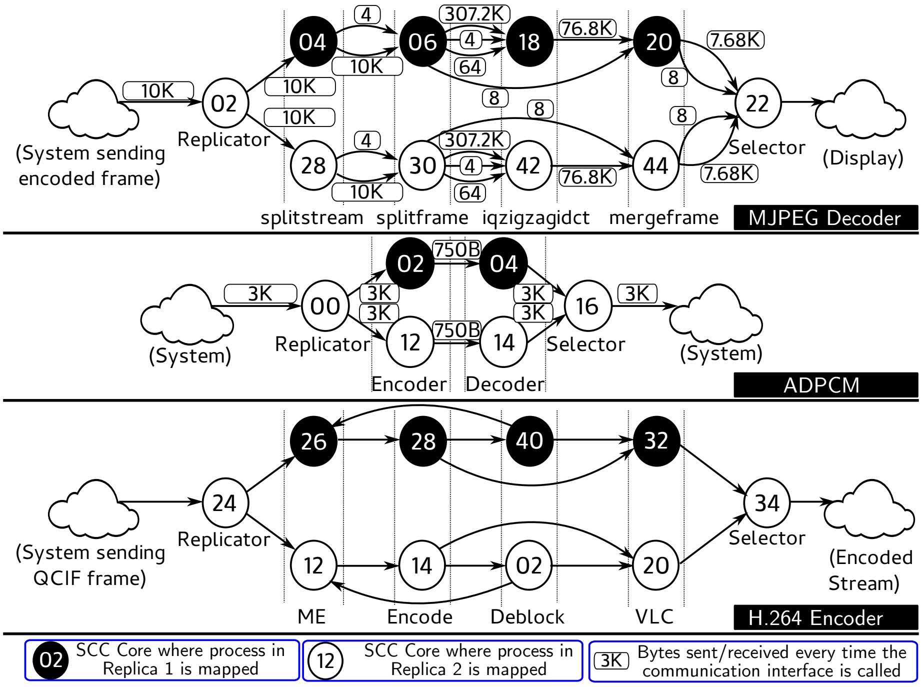


Figure 6.3: The MJPEG Decoder (top) and ADPCM Application (middle), and the H.264 applications.

<P,J,D> model, commonly used in real time systems. In case of a fault, the faulty replica stops producing (or consuming) tokens altogether.

The MJPEG Decoder

For the fault tolerant MJPEG decoder, the input to the replicas is an encoded frame ($\sim$ 30 fps). The replicator channel duplicates each token and provides it to the splitstream process in each replica. The mergeframe process(s) provides decoded frames to the selector, 320x240 pixels each. A token at the replicator and the selector channel is one encoded and decoded frame of sizes 10KB and 76.8 KB respectively. Note that it is possible to reduce token sizes by restructuring the application: i.e., split input frames into parts, and split decoded frames into parts. However, such adjustments depend on the application and the fault-detection latency requirements and are independent of the fault tolerance framework itself. After 18,000 frames, timing faults were introduced into the duplicated network and fault detection times are reported over 20 such runs.

The ADPCM Application

The system provides one data sample to the replicator every of 3KB every ~ 6.3ms. Note that the decoder rate is specifically tuned for the SCC. The encoder performs a 4:1 compression, which is reverted by the decoder. A

Fault Tolerant MJPEG Decoder

Communication Network	
Bandwidth	500 - 833 KB/s
Input interface to the replicator	*<Period, Jitter, Min. Interarrival Distance>*
Frame interarrival timings	<30ms, 2ms, 30ms>
MJPEG$_1$ (Replica 1)	
Frame consumption timings	*<Period, Jitter, Min. Interarrival Distance>*
Frame production timings	<30ms, 5ms, 30ms>
MJPEG$_2$ (Replica 2)	
Frame consumption timings	*<Period, Jitter, Min. Interarrival Distance>*
Frame production timings	<30ms, 30ms, 30ms>
Input interface to the consumer	*<Period, Jitter, Min. Interarrival Distance>*
Frame consumption timings	<30ms, 2ms, 30ms>

Fault Tolerant ADPCM Decoder

Communication Network	
Bandwidth	500 - 833 KB/s
Input interface to the replicator	*<Period, Jitter, Min. Interarrival Distance>*
Sample interarrival timings	<6.3ms, 2ms, 6.3ms>
ADPCM$_1$ (Replica 1)	*<Period, Jitter, Min. Interarrival Distance>*
Sample consumption timings	<6.3ms, 3.1ms, 6.3ms>
Sample production timings	<6.3ms, 7.5ms, 6.3ms>
ADPCM$_2$ (Replica 2)	*<Period, Jitter, Min. Interarrival Distance>*
Sample consumption timings	<6.3ms, 3.1ms, 6.3ms>
Sample production timings	<6.3ms, 12.3ms, 6.3ms>
Input interface to the consumer	*<Period, Jitter, Min. Interarrival Distance>*
Sample consumption timings	<6.3ms, 2ms, 6.3ms>

Fault Tolerant H.264 Encoder

Communication Network	
Bandwidth	500 - 833 KB/s
Input interface to the replicator	*<Period, Jitter, Min. Interarrival Distance>*
Frame interarrival timings	<30ms, 1.6ms, 30ms>
H.264 Encoder$_1$ (Replica 1)	*<Period, Jitter, Min. Interarrival Distance>*
Frame consumption timings	<30ms, 1.6ms, 30ms>
Frame production timings	<30ms, 1.6ms, 30ms>
H.264 Encoder$_2$ (Replica 2)	*<Period, Jitter, Min. Interarrival Distance>*
Frame consumption timings	<30ms, 10ms, 30ms>
Frame production timings	<30ms, 10ms, 30ms>
Input interface to the consumer	*<Period, Jitter, Min. Interarrival Distance>*
Frame consumption timings	<30ms, 1.6ms, 30ms>

Figure 6.4: Parameters for Fault Tolerance Experiments

token at both the selector and the replicator is one data sample of size 3KB. After 20,000 samples, faults were introduced in the ADPCM network, and fault detection times for 20 such runs are summarized.

The H.264 Encoder

The input to the H.264 fault tolerant encoder is a media stream in the *Quarter Common Intermediate Format* (QCIF) implementing the *Baseline Profile* with a resolution of 176x144 pixels, see [Kat13]. The encoder outputs the 4:1 compressed bitstream into *Network Abstraction Layer* (NAL) units. A token at the replicator is of size 840 bytes, whereas a token at the selector is 3KB in size. Note that the encoded NAL unit is not always 3KB is size, in which case the selector must first completely receive the full NAL unit from the given VLC replica. This is also the reason that performance at the selector is measured in terms of inter-token arrival times, as opposed to measuring encoded inter-frame timings. After 20,000 frames, faults were introduced in the H.264 encoder, and fault detection times while encoding 10 different inputs files are summarized.

6.6.1 Evaluation of the Framework

The framework described in this chapter is evaluated on the basis of (a) runtime overhead of the framework, (b) memory overhead of the framework (c) fault detection latencies and (d) comparison to distance function fault detection approach, see [NMA$^+$12].

Results

For all duplicated process networks, results in Table 6.5 show that under fault free conditions, the observed maximum number of *tokens* in various FIFO buffers is below theoretically computed capacities (*Theoretical Capacity* vs. *Max. Observed Fill*) validating the calculations presented in Section 6.4.4. The framework is extremely light, in both runtime and memory overhead. For example, the memory overhead in the case of the duplicated MJPEG decoder is 0.7% and 0.5% of the application code at the replicator and the selector channel respectively (excluding token storage, which depends on the application). The corresponding time overhead is at most 0.02% of the decoder inter-frame period. The overhead is found to be practically small enough that the duplicated and reference process networks can provide similar runtime performance. For example, for the MJPEG decoder, the decoded frame rate is almost identical (differences due to runtime overhead are in the order of microseconds) for both the reference and the duplicated process networks. Similar results hold for other applications. The framework detects faults within the bounds computed in Section 6.4.4, as can be seen by comparing fault detection latency statistics for each application vs. the computed upper bound. For instance, for the MJPEG decoder, the maximum latency for detecting a fault was found to be 103ms at the replicator channel, well within the computed upper bound of 180ms. Similarly, the maximum fault detection latency at the selector channel was found to be 102ms against the expected upper bound of 180ms. Notice that the in practical situations (i.e., in the experiments), the actual faults are detected much faster than the computed worst case bounds, since worst cases are only rarely encountered. Notice that the upper bounds for fault detection latency are not always symmetrical (e.g.,

FIFO	$\lvert\mathcal{R}_1\rvert$	$\lvert\mathcal{R}_2\rvert$	$\lvert\mathcal{S}_1\rvert$	$\lvert\mathcal{S}_2\rvert$	$\lvert\mathcal{S}_1\rvert_0$	$\lvert\mathcal{S}_2\rvert_0$
Theoretical Capacity	Capacities (tokens)				Initial tokens	
	2	3	4	6	2	3
Max. Observed fill (No Faults, 20 runs)	1	3	1	1	–	–

Fault Detection Latency	At Selector (ms)			At Replicator (ms)		
			Upper Bound			Upper Bound
	Min	99	180	Min	99	180
	Max	103		Max	102	
	Mean	100		Mean	100	

Overhead	Selector	Replicator
Memory	2.1KB+10Tokens (0.7%)	1.5KB+5Tokens (0.5%)
Runtime	7µs (0.02%)	2.9µs (0.01%)

Decoded Inter-Frame Timings	Reference(ms)	Duplicated(ms)
Min	29	29
Max	43	43
Mean	30	30

MJPEG Decoder

FIFO	$\lvert\mathcal{R}_1\rvert$	$\lvert\mathcal{R}_2\rvert$	$\lvert\mathcal{S}_1\rvert$	$\lvert\mathcal{S}_2\rvert$	$\lvert\mathcal{S}_1\rvert_0$	$\lvert\mathcal{S}_2\rvert_0$
Theoretical Capacity	Capacities (tokens)				Initial tokens	
	2	4	4	8	2	4
Max. Observed fill (No Faults, 20 runs)	1	3	1	1	–	–

Fault Detection Latency	At Selector (ms)			At Replicator (ms)		
			Upper Bound			Upper Bound
	Min	31	69	Min	26	69
	Max	38		Max	40	
	Mean	33		Mean	34	

Overhead	Selector	Replicator
Memory	2.1KB+12Tokens (6%)	1.5KB+6Tokens (4.6%)
Runtime	7µs (0.1%)	2.9µs (0.05%)

Decoded Inter-Frame Timings	Reference(ms)	Duplicated(ms)
Min	4.70	4.71
Max	8.25	8.25
Mean	6.18	6.18

ADPCM

FIFO	$\lvert\mathcal{R}_1\rvert$	$\lvert\mathcal{R}_2\rvert$	$\lvert\mathcal{S}_1\rvert$	$\lvert\mathcal{S}_2\rvert$	$\lvert\mathcal{S}_1\rvert_0$	$\lvert\mathcal{S}_2\rvert_0$
Theoretical Capacity	Capacities (tokens)				Initial tokens	
	2	3	4	4	2	2
Max. Observed fill (No Faults, 20 runs)	1	2	1	2	–	–

Fault Detection Latency	At Selector (ms)			At Replicator (ms)		
			Upper Bound			Upper Bound
	Min	91	120	Min	73	180
	Max	96		Max	88	
	Mean	86		Mean	81	

Overhead	Selector	Replicator
Memory	2.1KB+8Tokens (0.4%)	1.5KB+5Tokens (0.75%)
Runtime	7µs (0.02%)	2.9µs (0.01%)

Encoded Inter-Token Timings	Reference(ms)	Duplicated(ms)
Min	31	31
Max	40	40
Mean	35	35

H.264 Encoder

Figure 6.5: Results for the MJPEG, ADPCM, and H.264 Applications

the H.264 application). Also note that the selector and the replicator can *independently* detect faulty replicas as proposed in the chapter.

Brief Comparison to the State-of-the-Art

Since the distance function approach is superior than simple watchdog approach, we restrict our comparison to distance function approach in this chapter. Distance functions monitor the fault status of an input (or output) stream by verifying whether or not the timestamps of received tokens fit an expected distance function. Since in our framework both the replicator and the selector can monitor faults in the replicas, we modify the distance function such that:

Application	Fault Detection Latency(ms)					
	Distance Function Approach			Our Approach		
	Max	*Min*	*Mean*	*Max*	*Min*	*Mean*
`MJPEG Decodper`	48.2	48.1	48.1	47.1	47.0	47.0
`ADPCM Application`	7.3	7.1	7.2	6.3	6.3	6.3
`H.264 Encoder`	31.4	31.2	31.3	30.4	30.1	30.3

Table 6.1: Comparison of our proposed approach with distance function approach.

- The modified distance function can monitor faults at the replicator *as well as* the selector, and
- The modified distance function can operate with the fail-silent fault model assumed in this chapter

In order to avoid any false positives from the *l* distance approach, the timing variations from the replicas are minimized, allowing the expected distance function to be constructed with $l = 1$. This way, we compare the *minimum* fault detection timings from both approaches. The fault detector at the replicator observes the timings at which the replicas consume the tokens by monitoring the respective FIFO buffers at the replicator channel. To this end, a 1ms timer was introduced, one for each replica (as described in the original distance function chapter) which allows the detector to monitor the status of both FIFO buffers with a resolution of 1ms. Since the baremetal SCC environment does not provide any threading support, we used the iRCCE non-blocking communication library in order to ensure that the fault detector itself does not block when the replicator blocks on the FIFO read or write interface, see [RSS$^+$13]. To this end, the replicator channel was modified to poll both FIFO buffers every 1ms, using the threading technique developed for the baremetal Intel SCC, see [RSS$^+$13]. The choice of 1ms polling interval reflects the standard tick resolution on most *nix based computers. This setup allows the observer to declare the replica as faulty when it (a) slows down beyond the acceptable rate, and (b) stops silently.

Furthermore, for a fair comparison, all applications were tuned so that jitter and variable delays were minimized as much as possible. This avoids the necessity of first approximating the distance function to an *l-repetitive* distance function, which may lead to false positives.

The results comparing fault detection latencies at the replicator are summarized in Table 6.1. The fault detection latencies at the selector are similar, and therefore, are not shown.

Brief Discussion

Note that the fault detection latencies using the detection approach is always greater than our method. This is solely due to the choice of having a 1ms polling interval and having non-integer application periods (e.g., 6.3ms for the ADPCM application). In principle, it is possible to set the polling interval at a finer granularity, but at the cost higher resource overhead. In summary, compared to the proposed technique, the obvious additional expenses imposed by the l distance approach are the timer resources. Furthermore, the l-distance function approach is limited to a subset of general `<P,J,D>` model, and optimizations for compute and memory resources requires further approximation of the distance function, raising the possibility of false positives. However, in many use cases, such approximations can be safely done, in which case, the l distance approach is equivalent in performance to the proposed approach, except for additional requirement on timer resources.

6.7 Closing Remarks

The overall vision is that the system designer takes a multi-pronged approach to the design and implementation of a reliable multiprocessing system. First, the system is designed in order to avoid one or more class of faults (e.g., thermal faults). Incorporating appropriate design measures to avoid a wide class of faults may be expensive, in terms of design and analysis effort required, or resources required to implement such a design. Therefore, it may be more attractive in terms of the overall resource efficiency to simply tolerate other faults, which can be done based on the technique presented in this chapter.

To this end, this chapter presented an efficient (i.e., memory and runtime overhead) arbitration logic (the selector and the replicator channels) together with simple timing fault-detection strategies to construct a fault-tolerant real time process network. It was shown that the fault tolerant network is equivalent in functionality and timing to the original process network it was derived from. The proposed approach relies only on the timing properties of the interfaces of various components (e.g., replicator, selector, and the vulnerable system) thus making it appropriate for use with complex or safety critical applications wherein a re-design of the application may not be feasible. Experiments on the state-of-the-art many core Intel SCC processor validate the proposed technique and efficiency claims.

7

Recovering from Faults in Process Networks

Summary

This chapter presents a new technique to recover from faults in a multiprocessing system. The technique proposed in this chapter is restricted to scenarios in which a system does not fail immediately after having experienced a fault. Furthermore, the proposed fault recovery technique can be used alongside fault tolerance technique proposed in the previous chapter, thereby potentially achieving better resource efficiency while providing a reliable system.

7.1 Introduction

This chapter presents a new technique that can be used to recover from faults in a multiprocessing system. The fault recovery mechanism proposed here complements the fault tolerance approach described in the previous chapter. The overall idea is to design a system which is able to avoid specific class(es) of faults, such as thermally induced faults, but has the capability to tolerate a broad class of faults, such as those caused by hardware malfunction. The previous chapter proposed a minimum of $n = 2$ simultaneously executing replicas of the critical application (i.e., the application, or part thereof which must be made fault tolerant) which can tolerate at most $n-1$ permanent faults at a time, see Chapter 6. In principle, one may implement a system with a very large number of simultaneously executing replicas of the critical application (i.e., $n >> 1$) which will be able to tolerate an arbitrarily large number of faults. However, this may not necessarily be the best technique. In fact, such an approach may make the system more vulnerable to faults as compared the system which executes $n = 1$ replicas of the critical application. For example, if the critical application is compute intensive, and therefore, heats the core on which it executes, then having too many replicas of the same application may cause the processor to overheat, reducing the reliability of the system. Based on the expected arrival rate of faults into the system, it may be better to have only a few (e.g., $n = 2$) simultaneously executing replicas of the critical application, and should a fault occur, recover from the fault transparently to the observer. The fault recovery happens in parallel to fault tolerance, and therefore, can achieve better efficiency in terms of resource usage while still ensuring a reliable system.

The fault recovery technique proposed in this chapter is restricted to those classes of faults wherein the system does not fail immediately after having experienced a fault. An example is a thermal fault, wherein a system (e.g., a core in a multiprocessor) is said to have experienced a thermal fault if the temperature of that core exceeds a pre-determined threshold, but the core remains operational until its temperature exceeds the critical temperature. The limited time between the instants when the system experiences a fault and ultimately fails is utilized to migrate the critical application from a faulty (e.g., overheated core) to a new, cooler location. Beyond fault tolerance, the technique proposed in this chapter can also be used for dynamically balance workload in a multicore or a many-core processor.

Migration requires that upon detecting an event, an application can be brought to a *stable state* where all processes involved in migration have stopped executing, collected all data packets sent to them, do not send any new data, and all local variables have been saved, including the program counters. Only when such a state is reached, contexts can be saved correctly, applications (or parts of them) can be migrated, and safely restarted from the point where these had been interrupted.

Bringing an application (or parts of it) to a stable state is not trivial when the application is distributed, composed of many asynchronously executing processes which do not share clocks or memory, with possibly asynchronous communication, and no prior knowledge of the amount of data being produced (or consumed) by a process in any given interval of time, e.g., applications following Kahn Process Network semantics.The model is quite often used for specification and design of control and signal-processing applications which are ubiquitous today.

Before proceeding, we summarize the problem considered in this chapter:

Given a process network executing on a distributed memory system, upon the detection of an event, the process network needs to be brought to *a stable state which is suitable for the migrating all or part of the given process network. The technique should be lightweight, safe, correct, and work independently of the timing properties of the system or the topology of the process network.*

To this end, this chapter proposes a technique which efficiently and correctly brings a process network executing on a distributed system to a known *stable state*. The correctness of the technique is independent of the temporal characteristics of the system and the topology of the process network. The required modifications of a process network are lightweight and preserve its original functionality. A model characterizing the timing properties of the technique is provided. The feasibility and efficiency of the proposed approach and the respective model are validated with experimental results on Intel's SCC platform.

7.2 Related Work

Lots of research results have been published on process migration techniques, see e.g. [LKP$^+$10, B$^+$06, A$^+$09, A$^+$08], however, these usually target shared memory systems or do not provide any details on how to guide a general process network to a stable state where contexts can be saved correctly. Moreover, timing models are rarely provided or discussed. Similarly, this is the case for load-balancing literature [LL94, RW89, SW92].

A process migration technique for Polyhedral Process Networks (PPNs) has been proposed in [C$^+$12]. PPNs are a restricted form of KPNs since all loop bounds, array indices, and index expressions must be affine expressions and a process cannot change these parameters at run-time. Therefore, the technique targeting PPNs is not applicable to general KPNs. In the proposed technique, a process execution can be stopped at any time. However, this may require re-execution of the same code after migration which is in contrast to the proposed solution that does not require re-executions. Moreover, the PPN approach relies on a complex middle-ware system that continues to run on the processing node even if the application is migrated making the technique unsuitable in cases the reason for migration is high temperature. In contrast, in our approach, the affected core can completely stop after a known time, called the *stabilization time*. This is in contrast to the PPN work which does not discuss memory requirements, timing properties, or the correctness of the proposed stabilization technique.

Kernel-based approaches for process migration usually require the usage of specific features of an operating system or modifications to the OS kernel making them non-portable, e.g. [MC02, C$^+$06]. In this chapter, we focus on solutions that work in user-space which do not depend on any specific operating system features, guaranteeing the portability of the solution.

Checkpointing provides a means to manage the context of a migrating process, e.g., the Berkeley Labs Checkpointing and Restore (BLCR) algorithm [S$^+$05]. However, checkpointing requires a fairly complex bookkeeping process where all processes must log all incoming tokens, all calculations, and all output tokens between each checkpoint. Thus, checkpointing can easily overwhelm the computational capabilities of a typical embedded system [W$^+$10]. In this chapter, we focus on a *lightweight* approach which avoids rolling back, but is able to bring the process network to a stable state which is ready for migration.

Chandy and Lamport [Cha85] have proposed an approach to taking snapshots of a process network on a distributed system which is similar to the stabilization problem that we handle. However, they restrict themselves to (theoretical) systems with infinite FIFO channels and rule out the possibility that a process may block when attempting to write on a full output channel. In contrast, our technique is applicable to practical systems with bounded FIFO channels. Furthermore, we provide an implementation and a timing model which are validated with experiments.

7.3 Motivational Examples

In this section, we illustrate the challenges involved in the stabilization of a process network executing on a distributed memory system by using two simple examples. As is usually assumed in the case of general dataflow process networks, and specifically Kahn Process Networks [Kah74], processes have blocking semantics, i.e., a process attempting to read from an empty input FIFO buffer will block. Likewise, a process which attempts to write data tokens to a full output FIFO buffer will also block. The overall challenge is the complexity involved in getting a process, or a set of processes to suspend their usual computation activities, stop transmitting any new data to other processes, and also ensure that each process collects all the tokens which have already been transmitted to it by other processes. Furthermore, a process must enter the stable state, i.e., suspend its computation and communication activities, on a cue, such as reception of a distinguished signal

Example 1: Decentralized Stabilization. Consider the process network shown in Figure 7.1(a), with three processes v_1, v_2, and v_3 and the possibility that v_1 and v_2 must enter the stable state in order to migrate. In this example, assume that a process can autonomously decide whether or not it must prepare to migrate (i.e., enter the stable state) by sensing its environment.

Assume that v_1 is successful in suspending its computation, and therefore stops transmitting any data to v_3. However, v_3 has not yet decided to enter the stable state, and continues to expect data tokens from v_1. Since v_1 has stopped sending data tokens to v_3, the latter will eventually block when it attempts to read from the input FIFO $\mathcal{F}(v_3, 1)$ due to insufficient

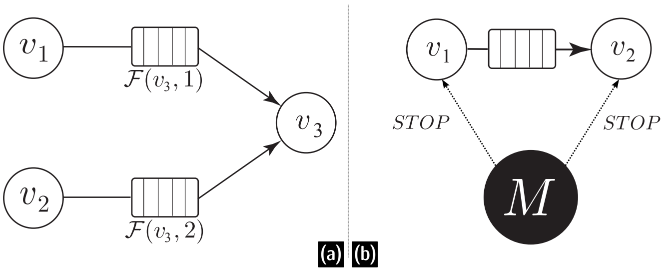


Figure 7.1: Motivation examples. Left: fully distributed stabilization approach. Right: Fully centralized stabilization approach.

number of tokens. Since v_3 is now blocked, it ceases to read any tokens from the FIFO $\mathcal{F}(v_3, 2)$, which causes it to fill up. Thus, when v_2 attempts to write a new token to the full FIFO buffer $\mathcal{F}(v_3, 2)$, it gets blocked. The entire system is now deadlocked. Consequently, should v_2 now decide to enter the stable state, it will not be able to do so. In principle, the process v_3 can be unblocked when the process v_1 resumes normal operation after migration, allowing v_2 to enter the stable state. However, such an approach requires a external entity (such as a *master process*) to co-ordinate migration procedure of each process in the network. The master process must carefully analyze dependencies (i.e., determine each process' parents and children), and use this information to schedule each process' progress into the stable state.

Example 2: Coordinated Stabilization. For this example, assume that a separate master process is connected to all processes via event channels. The master process initiates the stabilization of the processes by sending a `stop` event. Consider the producer (v_1) and consumer (v_2) process network shown in Figure 7.1(b). Once the affected processes receive the `stop` event, they can start the stabilization procedure.

Suppose that the master sends the `stop` event to both v_1 and v_2. Let v_2 receive the event before v_1. Stabilization requires that v_2 has received all the data tokens which have been transmitted to it by its parents, including all those tokens which may be in flight. Furthermore, v_2 must cease all computational activities, and must not transmit any more data tokens to its children. Although v_2 has received the `stop` event, it cannot simply cease operating since it has no way of determining whether all tokens transmitted to it by v_1 have been received. On the other hand, v_1 is unaware that v_2 will be entering the stable state, and keeps transmitting new data to it, thus preventing v_2 from stabilizing. Furthermore, should v_2 simply cease to read data tokens from its input FIFO buffer, it will eventually fill up, causing v_1 to block. Also, if v_2 drops any data tokens which continuously arrive over its input channels, then the execution context for v_2 cannot be calculated correctly and tokens may be lost.

The example shows that simply having a master process which tries to bring all processes to a stable state is not sufficient. Due to different network latencies, processes may go into stable states at different times. Therefore, the correctness of the coordination technique needs to be time insensitive. Further, a mechanism is needed which indicates when all processes have stopped and special tokens have to be sent to child processes to notify that the last data token has arrived.

7.4 Model and Definitions

A process network $\mathcal{N}$ is defined as the tuple $\mathcal{N} = (V, C, I, O, F, i, o, c, f)$, where V is a set of processes, C is a set of data channels, I is a set of input ports, O is a set of output ports, and F is a set of bounded first-in first-out (FIFO) buffers. The function $i : V \to \mathcal{P}(I)$ maps a process to a set of input ports, where $\mathcal{P}(S)$ denotes the power set of a set S. The function $o : V \to \mathcal{P}(O)$ maps a process to a set of output ports. Processes cannot share input or output ports, i.e., $i(v_k) \cap i(v_j) = \emptyset$ and $o(v_k) \cap o(v_j) = \emptyset$ for all $v_k, v_j \in V, v_k \neq v_j$. A process $v \in V$ reads data tokens from its input ports $i(v)$ and writes data tokens to its output ports $o(v)$. The function $c : U \to C$, where $U = \{(a, b) : a \in o(v_k), b \in i(v_j), v_k, v_j \in V, v_k \neq v_j\}$, maps pairs of output and input ports, belonging to different processes, to a data channel.

The function $par : V \to \mathcal{P}(V)$ returns the set of parent processes for a given process, and the function $ch : V \to \mathcal{P}(V)$ returns the set of child processes for a given process.

The function $f : V \times I \to F$ provides a FIFO buffer $\mathcal{F}(v, m)$ to an input port m of a process v. The buffer has a finite size denoted as $|\mathcal{F}(v, m)|$. A process attempting to write to an output port connected to an input port with a full FIFO will block until there is sufficient space available. Similarly, a process attempting to read data tokens from an input port with an empty FIFO will block until there are sufficient tokens available. Conventionally, KPNs [Kah74] assume unbounded FIFOs, however, practical systems with finite resources impose maximum sizes on the FIFOs [N$^+$08, H$^+$09, GB10]. Therefore, the notation adheres more closely to the typical implementation of a process network.

Bringing a process network $\mathcal{N}$ into a state that is ready for migration requires that each process $v \in V$ reaches a stable state which is defined as follows:

Def. 7.1. *(STABLE STATE) A process $v \in V$ enters a stable state if it does not perform any more computations, parents $par(v)$ do not send any new data tokens, v does not send any new data tokens to its children $ch(v)$, and it has received all data tokens already sent by its parents $par(v)$.*

Once such a stable state has been reached, the context of each process can be saved and migrated. The context of a process includes all unread tokens (all not yet processed input tokens), all produced but unsent tokens, and the program state. Given this context, a process can be safely restarted.

Having the ability to bring each process to a stable state may require that the original processes are slightly modified. Such modifications should preserve the correct functionality of the network. Thus, it must be ensured that the original process network $\mathcal{N}$ is functionally equivalent to the modified process network $\mathcal{N}'$. In other words, the solution must comply with the notion of *Correctness* defined as:

Def. 7.2. *(CORRECTNESS) Given two process networks $\mathcal{N}$ and $\mathcal{N}'$, where $\mathcal{N}'$ is a modified version of $\mathcal{N}$ such that it has mechanisms to be brought to a stable state. We say that $\mathcal{N}'$ is correct, if for any process $v' \in V'$ of $\mathcal{N}'$ which corresponds to process $v \in V$ of $\mathcal{N}$, for any vector of input sequences of data tokens In, the following relationship holds:*

$$In \xrightarrow[(v \in V)]{} Out \implies In \xrightarrow[(v' \in V')]{} Out \tag{7.1}$$

where $In \xrightarrow[(v \in V)]{} Out$ means that process v produces the vector of output sequences of data tokens Out, when provided with the vector of input sequences of data tokens In.

Thus, the overall problem of this chapter can be summarized as: Extend the process network $\mathcal{N}$ to $\mathcal{N}'$, such that:

1. $\mathcal{N}'$ is functionally equivalent to $\mathcal{N}$;
2. $\mathcal{N}'$ can be brought into a stable state independent of computation or communication delays.

7.5 Proposed Technique

Interruption of the normal execution of a process network is initialized and coordinated by a central authority, which can be either an external process, a process of the existing network, including the process which requires migration. Without loss of generality, we assume that the central authority is an external process called the "master".

We start by defining a set of coordination events which will be used by the master for the communication with all processes. The set of coordination events $\mathcal{E}$ contains the `stop` event: a process receiving this event will *eventually* stop any computations and data transmissions to children, and must then acknowledge the reception of the event; and the `proceed` event: a process receiving this event must proceed to collect its context. The master sends the `proceed` event only when it has received all acknowledgments for the `stop` events.

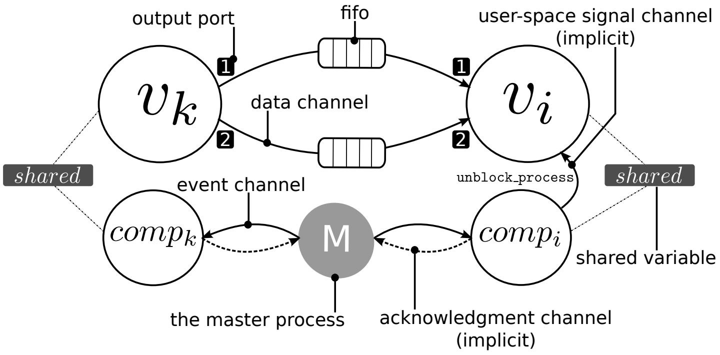


Figure 7.2: Process, companion process, master, event channel, and signal.

In this chapter we focus on stabilizing the entire process network, therefore, the master process always broadcasts the `stop` event to all processes in the network. However, the algorithm can be easily extended to stop only specific processes in the network, see Section 7.6. Furthermore, a prototype implementation of the proposed stabilization technique is discussed in Section 7.7.

The master uses a (bidirectional) event channel $e_j \in E$ to communicate with process v_j. If the process is blocked because of reading from an empty FIFO or writing to a full FIFO, it will not be able to detect (and process) events from the master, therefore the event channel $e_j \in E$ is not directly connected to process v_j, but to a companion process $comp_j$, as shown in Figure 7.2. The companion process $comp_j$ is very lightweight. It only receives and processes events from the master and makes them available to its process via a shared variable $shared$. When $comp_j$ receives the `stop` event, it sets the shared variable $shared$ to `stop` and sends a signal `unblock_process` to v_j that cancels any blocking read or write of process v_j.

Process v_j checks the variable $shared$ at the beginning of each communication primitive (a read or write statement) and exits normal execution if $shared$ is set to `stop`. If process v_j is blocked and receives the signal `unblock_process`, it also exits normal execution, otherwise the signal has no effect. Once the process exits execution, it sets the shared variable $shared$ to `done`, and then, the companion process can send an acknowledgment back to the master.

The above mechanism is only a conceptual description of our technique. The actual implementation of a separate companion process or unblocking signals will depend on the underlying platform.

7.5.1 Collecting Data Tokens

When a process exits normal execution, it executes a special function called $Wrapup$. Here no more data transmissions or computations are performed. First, the function waits until the process has received the `proceed` event from the master, i.e., meaning that all processes have suspended their normal computational activity. Then, it collects the process context and performs any other housekeeping steps such as returning any allocated memory.

During the collection of the context, a process must collect all tokens that are sent by its parents. If these tokens are not collected, the data tokens are "lost", which leads to incorrect behavior. This is further complicated by the fact that there might be a number of tokens arriving late due to late arrival of `stop` events in parent processes. Therefore, it must be ensured that there are no data tokens which are "in flight", i.e., written by a parent process but not yet received in the local FIFO of the child process. Otherwise, the technique would not be delay-independent.

In order to remedy this problem, in the $Wrapup$ function, each process, after receiving the `proceed` event, sends an end-of-stream (EOS) token to all its children. Thus, a process must continue to collect late arriving tokens from its input channels until the EOS marker has been received on each channel.

7.5.2 Bounding the Size of Contexts

For the purpose of discussion, the FIFO $\mathcal{F}_v(m)$ for an input port m of process v is divided into two FIFOs: $\mathcal{M}_v(m)$ and $\mathcal{L}_v(m)$. The size of $\mathcal{F}_v(m)$ is the sum of sizes of $\mathcal{M}_v(m)$ and $\mathcal{L}_v(m)$. Tokens move from $\mathcal{M}_v(m)$ to $\mathcal{L}_v(m)$ when there is sufficient space in $\mathcal{L}_v(m)$. The separation is made in order to reflect more closely real implementations of process networks, where $\mathcal{M}_v(m)$ refers to buffers of the interprocess communication layers and the capacity of communication links, while $\mathcal{L}_v(m)$ refers to the FIFO local to a process. The process has only the knowledge of the current status of $\mathcal{L}$ but not of $\mathcal{M}$.

We assume that the number of late-arriving tokens to each process can be bounded. This is the case for any NoC-based communication where the network capacity can be statically calculated (or at least upper bounded) by analyzing its topology, and it is the case for many communication libraries where the buffer sizes of data links are finite.

The memory space to absorb all late-arriving tokens is provided by a statically allocated set of "backup FIFOs" $\mathcal{B}$, in particular, one for each input port FIFO. The maximum number of late arriving tokens that a process v must absorb on each input port m and store in a backup FIFO is upper bounded by: $|\mathcal{B}_v(m)| = |\mathcal{M}_v(m)| + |\texttt{EOS}|$ where $|\mathcal{M}_v(m)|$ denotes the size of FIFO $\mathcal{M}_v(m)$, and $|\texttt{EOS}|$ denotes the size of the EOS marker. The bound is correct even if the local FIFO $\mathcal{L}_v(m)$ is full, since tokens in the input FIFOs are not considered as late arriving tokens and therefore not saved in the backup FIFOs. The backup FIFOs $\mathcal{B}$ are not available during the normal course of operation. Upon reception of the `proceed` event and before sending the EOS tokens, a process v_j swaps all regular FIFOs $\mathcal{L}_{v_j}(m)$ with the corresponding backup FIFOs $\mathcal{B}_{v_j}(m)$.

Consequently, an upper bound on the size of the context of process v is:

$$D_v^* = \sum_{\forall j} \{|\mathcal{L}_v(j)| + |\mathcal{B}_v(j)|\} + |\mathrm{LN}| + |\mathrm{LV}| \tag{7.2}$$

where |LN| is the memory space required to store the line number of the program when v exited the normal execution, and |LV| is the memory space required to store all local variables (loop indexes, unsent tokens, etc.).

Note that many existing solutions for migration do not rely on backup FIFOs but simply use a constantly running middle-ware system that will re-direct any late arriving data, no matter how late it is. However, such solutions are not always feasible if, for example, a processing node is close to reaching peak temperature and any processing activity on it needs to be stopped after a certain time.

7.5.3 Timing Analysis

The proposed algorithm to stabilize a process network is delay-independent. However, in case that the maximum time to transmit a token between two processing nodes and the maximum time that a process is executing without calling a communication primitive are known, an upper bound on the overall stabilization time for a process network can be calculated. Such timing parameters can be obtained either with formal analysis, in which case the computed bounds would be hard real-time ones, or by measurements (or simulations), which leads to soft real-time bounds.

In order to analyze the timing, we consider two phases of the algorithm. In the first one (denoted as $phase1$), the master (denoted as M) broadcasts the `stop` token to all processes and waits for all acknowledgments. In the second one (denoted as $phase2$), it broadcasts the `proceed` token and then each process will simply absorb all data tokens from its parents until it receives an `EOS` marker on all its input ports.

The maximum time between the time instant when the master broadcasts the `stop` token and the instance it receives the acknowledgment from process v is composed of four time periods: (i) the maximum time $t^*_{M \to v}$ for the `stop` token to travel from the processing node of the master to the one of process v, (ii) the maximum time $t^*_{read,v|write,v}$ that process v requires to perform a single read or write of a data packet of maximum size, (iii) the maximum time $t^*_{c,v}$ that process v is executing without calling a communication primitive, and (iv) the maximum time $t^*_{v \to M}$ for the `ack` token to travel from the processing node of process v to the one of the master.

In other words, the master receives the acknowledgement from process v no later than after the following time period:

$$t^*_{phase1,v} = t^*_{M \to v} + t^*_{read,v|write,v} + t^*_{c,v} + t^*_{v \to M} \tag{7.3}$$

and can broadcast the `proceed` token no later than after the following time period:

$$t^*_{phase1,\mathcal{N}} = \max_{v \in V} \left\{ t^*_{phase1,v} \right\}. \tag{7.4}$$

Afterwards, each process waits until it receives the `proceed` token, swaps all regular FIFOs with the backup FIFOs, and waits until it receives an `EOS` marker on each of its input ports. The amount of time between the instant when the master broadcasts the `proceed` token and the instant when the all targeted processes have entered the stable state is upper bounded by $t^*_{phase2,v}$, see (7.5). The bound $t^*_{phase2,v}$ is dependent on two scenarios: (i) a process v receives the `proceed` token *before* its parent(s) $u \in par(v)$ have received it, and therefore, must absorb all data tokens arriving at its input interface(s) until it has received the `EOS` marker from all its parents, or (ii) the parent(s) $u \in par(v)$ of a process v receive the `proceed` token before any of their children. In this case, a child v of u must continue to absorb all data tokens at its inputs[1] till it has received the `EOS` token from all its parents. It may be possible that v receives the `proceed` from the master *after* it has received the `EOS` from its parents, and therefore, in this scenario, the worst case time bound is independent of the time instant v receives the `proceed` token. In both cases, a process v must wait for the `EOS` token from all its parents, which makes it possible to derive a simple timing bound $t^*_{phase2,v}$ in (7.5).

$$t^*_{phase2,v} = \max_{\forall u \in par(v)} \left\{ t^*_{M \to u} + t^*_{u \to v} \right\} \tag{7.5}$$

where $t^*_{u \to v}$ is the maximum time for the `EOS` marker to travel from the parent process u to the one of its children, v. Note that this already includes the time required for all tokens transmitted by u to travel to v.

Finally, the stabilization time of process v and process network $\mathcal{N}$ is upper bounded by:

$$\begin{aligned} t^*_{stab,v} &= t^*_{phase1,\mathcal{N}} + t^*_{phase2,v} && (7.6)\\ t^*_{stab,\mathcal{N}} &= t^*_{phase1,\mathcal{N}} + \max_{v \in V} \left\{ t^*_{phase2,v} \right\}. && (7.7) \end{aligned}$$

It can be seen from (7.7) that the overall stabilization time depends on the time it takes for *phase 1* and *phase 2* to complete. The length of phase 1 is dominated by the maximum *process compute time*. The length of phase 2 is dominated by network speed, with time taken to swap FIFO being negligible. Furthermore, the stabilization time is (largely) independent of the amount of data being read or written by a processor, since blocking reads and writes activities of a process are canceled with immediate effect by a signal from the companion process, also see Algorithm 9, lines 16 - 19. The network speed accounts for a small part in the overall duration of the stabilization time, since a maximum of $|\mathcal{M}_v(m)|$ tokens need to be collected in phase 2 before the stabilization is complete.

---

[1] Do recall that this process has already stopped computing and transmitting data tokens.

Do notice that (7.5) does not consider the |EOS| send and receive times by a process individually. This is because the time it takes for a process v to receive the |EOS| token from its parent already covers the time it takes for v's parent to send the |EOS| tokens (plus a small communication time), and thus the use of $t^*_{u \rightarrow v}$.

The property of Correctness. The proposed technique requires three modifications of the process network, namely the addition of a companion process, a $Wrapup$ function, and a conditional check before proceeding with a blocking read or write. The addition of the companion process $comp_j$ to process v_j does not change the order of any tokens in any channel for any process. It only retains the information that an event $\{\texttt{stop}, \texttt{proceed}\} \in \mathcal{E}$ was dispatched from the master. The $Wrapup$ function simply stores late-arriving tokens in backup FIFOs, maintaining the relative order of arrival of tokens. Finally, the conditional check before proceeding with a blocking read or write does not interfere with computations or the tokens that are already read or need to be written. Thus, all three modifications preserve the original functionality of the process network and the correctness property is satisfied.

7.5.4 Summary of Distributed Process Network Stabilization Approach

A short summary of the approach is presented here:

1. The algorithm stabilizes the distributed process network *correctly*, i.e., no tokens are lost, and relative ordering amongst tokens in each channel is maintained. All tokens not consumed by a process v are stored in the process context in the correct order.

2. The stabilization procedure is composed of two phases, *phase 1* and *phase 2*.

    - Phase 1 brings all processes in the process network into a *known* state. The completion of phase 1 guarantees that no process in the network performs any further compute, write, or read steps. The length of phase 1 is dominated by *process compute times*.

    - Phase 2 required each process to swap local FIFO with backup FIFOs. The backup FIFOs are sized appropriately in order to accommodate all possible in-flight tokens from each of process' parents, plus the |EOS| token. Phase 2 is determined by the communication times, and hence dependent upon the network characteristics.

3. The algorithm is independent of the size of local FIFOs, and is the same as in the original process network. The size of backup FIFOs are statically determined.

7.6 Stabilizing Individual Processes

Our main section introduced an approach in which the entire process network was stabilized. However, the same principles can be applied to stabilize a part of the process network. Consider a process $v \in V$, which must stabilized. Therefore, all parents of v are informed of the stabilization of v, and as a result, process v receives the `EOS` from all its parents. The process v must also send `EOS` to all its children, so that v's children do not continue to expect tokens from the previous location of v. Once process v stabilizes, it can migrate. Subsequently, new channels must be established between the parents of v and v's children. Since the network structure is statically known, recreating channels is straightforward.

7.7 Implementing a Prototype

We now briefly outline a prototype implementation for the proposed stabilization mechanism. First, we start with the description of the structure of a process in the network (Section 7.7.1), which is expanded and adapted to enable a process to act on signals from its associated companion process (Section 7.7.2). The design of the companion process is described next (Section 7.7.3). We then follow up with the design of the *Wrapup* function which is responsible for computing the context of the associated process (Section 7.7.4). The section concludes with a brief note which describes *one* method which can be used to unblock a process based on a signal from the associated companion process, see Section 7.7.5.

7.7.1 Process Structure

We start with illustrating a high-level API for specifying process networks. A process $v \in V$ starts executing by first initializing itself at Line 2 in the Algorithm 8, and then repeatedly invokes the $Fire$ function at Line 4. The $Fire$ function implements the functionality of the given process, and can consist of any number of data-token read/write steps, and compute steps (e.g. branches, loops, assignments, etc.). Furthermore, the communication and computation steps can occur in any order in the $Fire$ function.

```
1 start process $v$
2     Init();                                  // Initialization
3     while true do
4         Fire();                // Communication and Computation
5     end
6 end
```

Algorithm 8: Basic structure of a process $v \in V$.

7.7.2 Integrating the Stabilization Mechanism

We now expand and adapt the structure of the process in Algorithm 8 to integrate the stabilization mechanism. To this end, we seek minimal changes to the original structure of the process so that a process gains the ability to go into the stable state and collect its context. The modifications discussed in Section 7.5 are incorporated into the pseudo-code shown in Algorithm 9. Line 11 checks for the `stop` event *before* starting a potentially blocking data token read or write step. If no event has been posted, the process starts the data-token read or write step. If the process is blocking on a read or write step while the `stop` event from the master is received, the signal `unblock_process` will unblock the process.

```
1  start process $v_j$
2      Init ();                                               // Initialization
3      while !cancelled do          // Until the stop event cancels further execution
4
5          cancelled ← Fire ();                    // Communication and Computation
6      end
7      Wrapup ();          // Call the Wrapup function to finish stabilization
9      Function Fire()
10         ...
11         if $shared == stop$ then // Do not start read/write step if $shared$ is stop
12
13             shared ← done;
14             return (cancelled ← true);
15         else
16             Start a blocking R/W step;
17             if unblocked by signal then
18                 shared ← done;
19                 return (cancelled ← true);
20             end
21         end
22         ...
23 end
```

Algorithm 9: New structure of a process v_j.

Unblocking upon reception of an event is easily accomplished by using user-space signals from threading libraries such as the POSIX library. Thus, the reception of the `stop` event by a process effectively cancels the currently blocked token-write or token-read operation.

7.7.3 The Companion Process

The companion process $comp_j$ of process v_j is responsible for the communication of process v_j with the master process. Algorithm 10 describes the companion process $comp_j$, which is executed independently of process v_j. The design implements a handshake between a companion process and the master In particular, the companion process waits for an event of the master. If it finds a `stop` event, it updates the $shared$ variable and waits until the variable is set to `done`. Afterwards, it sends an acknowledgement to the master and waits until it receives the `proceed` event.

```
start process comp_j
    while true do                          // Run as a separate concurrent process

        Read event from event channel;
        if found stop then
            shared ← stop;
        end
        Sleep on the shared variable until it is changed to done;
        Send acknowledgment to master;
        Read event from event channel;
        if found proceed then
            shared ← proceed;
        end
    end
end
```

Algorithm 10: Pseudo-code illustrating the functionality of the companion process $comp_j$.

It is possible to optimize this structure such that not every process in the process network has a companion process, but instead a group of processes residing on one processing element share a companion process. However, in the interest of simplicity, we do not explore such options in this chapter.

As a process v_j might be blocked because of reading from an empty FIFO or writing to a full FIFO channel, the companion process $comp_j$ has to be implemented as an additional object that is running in parallel to process v_j and just shares a single variable with process v_j. In case the platform supports multi-threading, the companion process $comp_j$ can be implemented as an additional thread. Otherwise, one can use stack-less threads as described in section 7.7.5. As the companion process is in a known state when the stabilization is completed, the companion process is not part of the context of process v_j, but can be re-initialized after migration.

7.7.4 The Wrapup Function

```
2 Function Wrapup()
3     Wait-to-Proceed ();                 // Wait for proceed event from master
4     Switch $\mathcal{L}(v_j, k)$ with $\mathcal{B}(v_j, k)$;
5     Forward-EOS () ;                    // Forward EOS token to all children
6     Collect-Tokens ();                  // Collect "late-arriving" tokens
7     Cleanup ();                         // Return memory to the system, etc.
```

Algorithm 11: Basic structure of the $Wrapup$ function.

The $Wrapup$ function was first introduced in Section 7.5.1. Pseudo-code illustrating the function is given in the Algorithm 11. First, in Line 3, it waits until it receives the `proceed` event from the master. It swaps all regular FIFOs $\mathcal{L}(v_j, m)$ with the corresponding backup FIFOs $\mathcal{B}(v_j, m)$ in Line 4. Afterwards, it sends an end-of-stream (EOS) token to all its children and waits in Line 6 until it receives an EOS marker from all its parents. Finally, some cleanup operations are performed to return the memory to the system.

Notice that swapping of the local FIFO $\mathcal{L}(v_j, m)$ with the backup FIFO $\mathcal{B}(v_j, m)$ preserves the correctness of the process network. This is because:

- A backup FIFO is brought online *only* after reception of the `proceed` signal. Note that a companion process transmits the acknowledgement only when the process changes the shared variable to `done`. Thereafter, the algorithm guarantees that the process will not transmit any more tokens. Therefore, in the worst case, the backup FIFO must be able to accommodate the tokens which are still in flight, which are upper bounded to $|\mathcal{M}(m)| + |\texttt{EOS}|$.

- The only token that a process transmits post-reception of the `proceed` event is the `EOS` marker, which is accommodated in the backup FIFO.

Therefore, it can be seen that (assuming that the communication network is lossless), none of the data tokens are lost in the process of stabilization. Further, the FIFO data structure maintains the relative ordering on the tokens on each channel.

7.7.5 Process Unblocking and User Space Context Management

The core problem at hand is to get the process to act on a message (or an event) immediately, which may require it to suspend its normal activity. There are two general approaches which are available to achieve such a behavior from the process. One approach requires that the process polls the event channel for a message (in this case, from the companion process), which can be easily implemented, but imposes a significant computing overhead as a process needs to check frequently for a rare event. The other approach is based on dispatching an unblocking signal to the process. This approach is also simple to implement, and usually requires the support of an operating system. Much of the complexity (e.g., dispatching a signal, waking up a process which receives the signal) is hidden from the user, and is managed by the operating system itself. Therefore, the actual solution used depends on a lot of factors, such as whether or not an operating system is available for use on the processor.

Irrespective of how a process is unblocked, it must then proceed to collect its context. This is a difficult problem, usually requiring interrupts and special operating systems, and may also also require specialized processors. In this chapter, we use stackless threads, such as protothreads for the purpose. Protothreads do not require any special operating system or hardware, and work completely in user space, see [DSVA06]. Protothreads have already been successfully applied to KPNs to provide lightweight scheduling [H$^+$09]. The functionality of protothreads is implemented as a set of macros that enclose the communication calls. The embedding of a KPN process into a protothread process can also be automated at the software synthesis step.

In a protothread, a control structure is used to store the local data of a process together with a variable that represents the line number of the process. Whenever the process exits the $Fire$ function, it updates this variable either to the current line number or, if the process has reached the end of the $Fire$ function, to the beginning of the $Fire$ function. On the other hand, at the beginning of the $Fire$ function, the line number variable of the control structure is read and the program counter jumps to this line. In order to extend the protothread library with the unblocking functionality, we change the process structure in two ways: First, we extend the `PT_WAIT_UNTIL` macro of protothreads with the abilities to check for the $shared$ variable and to be unblocked by the `unblock_process` signal, as outlined in Algorithm 9. Originally, the `PT_WAIT_UNTIL` macro just blocked a process until the read or write is successful. Second, we enclose each communication call with the extended `PT_WAIT_UNTIL` macro to obtain the functionality described in Algorithm 9.

7.8 Experiments

7.8.1 Overview

The feasibility and efficiency of the proposed stabilization technique are validated using two representative multiprocessing benchmarks: Demosaicing and a distributed Motion-JPEG (MJPEG) decoder algorithm, detailed in Section 7.8.4. Specifically, we measure the time required to bring the given applications into the stable state, and then compare the measurements with the corresponding theoretical upper bounds, see Section 6.6.

In order to achieve our goal, i.e., to compare the observed stabilization time with its upper bound, we proceed in three steps:

1. Calibration experiments are performed to derive a communication model of the target platform and to obtain the characteristics of the benchmark applications, see Section 7.8.3. As a result of this step, we calculate the upper bounds $t^*_{phase1,v}$, $t^*_{phase2,v}$, D^*_v for each process v, and $t^*_{stab,\mathcal{N}}$ for the network, see Section 7.5.
2. Stabilization experiments with the benchmarks are executed to observe the actual time taken by a process v to complete $phase1$ and $phase2$, the time $t_{stab,v}$ to stabilize, and the context size D_v.
3. The observed values are compared with the bounds calculated in Step 1.

7.8.2 Hardware Platform

For the experiments, we have used the Intel Single Chip Cloud (SCC) computer, with basic technical details having already presented in Section 6.6. In the context of this chapter, both benchmarks execute in the bare-metal configuration (i.e., no operating system on the processor) to reduce timing jitters. Cache-related timing variations are reduced by hosting one process per tile. Since the SCC implements a deterministic X-Y routing, timing variations due to router contention are reduced by carefully binding the processes onto the tiles. Inter-process communication is implemented using the iRCCE library [C$^+$11]. For timing measurements, all tiles establish a common time reference when they boot using the barrier operation available in the communication library. L2 caches and interrupts are disabled on all tiles. Data messages are at most of size $3\,\text{KB}$ each (longer ones are split) and control tokens are of size $16\,\text{B}$. The master process was placed on a separate tile so that it does not interfere with the application.

7.8.3 Calibration

The communication model was derived by observing the time taken to deliver a packet with size ranging from $4\,\text{B}$ through $3\,\text{KB}$ over hop distances ranging from one through eight. A total of 585 observations were made. The communication latency under high cross-traffic between any two processes u and v (including the master) mapped onto different tiles, was observed to be upper bounded by:

$$t^*_{u\to v} = 5.182|P| + 9935 \quad \text{(cl.cycles)} \tag{7.8}$$

where $|P|$ is the size of the payload in bytes. Because of the high cross-traffic, a dependency on the number of hops between the processing nodes of the communicating processes is not observed.

The calibration experiments indicate that the communication latency under high cross-traffic between any two processes u and v can be upper bounded. In particular, we have seen that the latency is independent of the number of hops between the processing nodes of the communicating processes. In addition, we observed the communication latency under low data traffic conditions, which is bounded by:

$$t^{\text{low traffic}}_{u\to v} = 5.182|P| + 307.4H + 531 \quad \text{(clock cycles)} \tag{7.9}$$

Here, the latency depends on the number of hops H between the nodes of the communicating processes. However, this model cannot be used as an upper bound as it considers only low data traffic in the network.

Another set of calibration experiments was performed in order to obtain the maximum computation time $t^*_{c,v}$ for each process. The detailed results of all calibration experiments are reported together with the detailed description of each benchmark in the following sections.

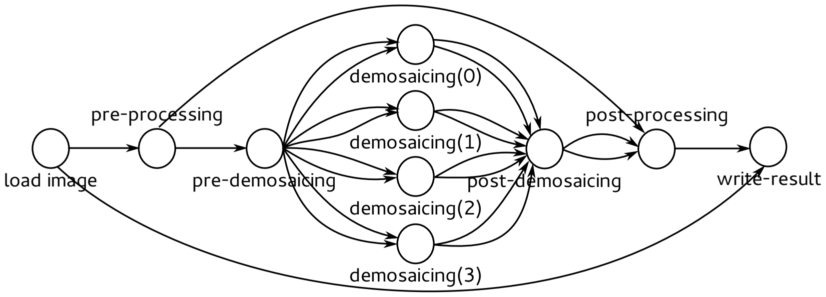


Figure 7.3: The Demosaicing application.

7.8.4 Benchmark Applications

Demosaicing. Demosaicing [Li05a] is both a compute and data intensive application consisting of 10 processes, see Figure 7.3. To measure the stabilization times, the experiment was repeated 20 times with different inputs and randomly varying the instants at which the master starts a `stop` token broadcast. Both the average and maximum values of the 20 runs are reported.

Table 7.1 summarizes the characteristics of the Demosaicing application obtained when measuring the computation and read/write times of the individual processes. The Demosaicing application was executed using five RAW images of different sizes, and the execution time for each different image was measured. The maximum execution time corresponds to the maximum time that a process executes without calling a communication primitive for each input RAW image. Finally, the maximum amount of data that is transmitted per outgoing channel and iteration is reported. The values given in Table 7.1 have been used in Table 7.2 to calculate the upper bounds on the stabilization times.

Table 7.2 compares the measured stabilization time $t_{stab,v}$ with the calculated upper bound $t^*_{stab,v}$. It can be seen that all processes did indeed stabilize before the expected time bounds. For some of the processes, the observed measurements are very close to the expected upper bounds. This means that the estimated bounds are indeed tight. The gaps between observed values and bounds are explained by the fact that $t^*_{stab,v}$ is mainly composed of the time the master waits until it receives all acknowledgments, and considers that a process can be in its longest computation section $t^*_{c,v}$. As shown in Table 7.2, $t^*_{c,v}$ is particularly large for the post processing process.

The overall context sizes reported in Table 7.2 is the sum of:

- The space required to store the line number (in order to restore the context), local variables (except those which store unsent output data tokens, and unread input data tokens), denoted as S_1.

- The space required for storing unsent output data tokens, and unread input data tokens, denoted as S_2.

process	binding	max. execution time	port	max. data / iteration
load image	26	0.13 s	0	618 KB
pre processing	12	0.13 s	0	618 KB
pre demosaicing	14	0.4 ms	1	160 KB
			2	2 B
			3	160 KB
			4	2 B
			5	160 KB
			6	2 B
			7	160 KB
			8	2 B
demosaicing_0	28	1.05 s	2	160 KB
			3	471 KB
demosaicing_1	18	1.07 s	2	160 KB
			3	471 KB
demosaicing_2	02	1.07 s	2	160 KB
			3	471 KB
demosaicing_3	40	1.05 s	2	160 KB
			3	471 KB
post demosaicing	30	2.4 s	8	618 KB
			9	618 KB
post processing	20	4.2 s	2	618 KB
write result	22	0.5 ms		

Table 7.1: Characteristics of the Demosaicing application. *Binding* refers to the core of the SCC on which the process executes.

Therefore, S_2 can be considered as the application-dependent context storage requirement, while S_1 is largely independent of the application. The contribution due to S_1 in the overall context sizes reported in Table 7.2 is presented in Table 7.3. It is clear from Table 7.3 that the total size of the context is dominated by the nature of the application itself.

In addition, the maximum measured size of the context D_v is compared in Table 7.2 with its upper bound D_v^* calculated using (7.2). For some of the processes, the observed measurements are equal to the estimated upper bounds indicating that (7.2) is tight. For some processes, the theoretical upper bound on the context size is about three times larger than the measured maximum size. The former assumes that all FIFO channels are full when the context is calculated, but in practice, some of the channels are only partly filled.

MJPEG Decoder. The second example is a parallelized version of the MJPEG decoder application taken from the benchmark suite of the Artist Network of Excellence [Art08]. The application consists of six processes and its structure is outlined in Figure 7.4.

Next, the characteristics of the MJPEG decoder application are summarized in Table 7.5. The following values are shown: The identification number of the SCC core, on which the process is executed; the maximum

process	$t_{phase1,v}$		$t^*_{phase1,v}$	$t_{phase2,v}$		$t^*_{phase2,v}$	$t_{stab,v}$		$t^*_{stab,v}$	D_v	D^*_v
	avg	**max**		**avg**	**max**		**avg**	**max**		**max**	
load image	0.024 s	0.13 s	0.13 s	19.46 μs	21.39 μs	29.33 μs	0.77 s	3.26 s	4.20 s	52 B	52 B
pre processing	0.042 s	0.16 s	0.16 s	40.80 μs	44.39 μs	48.12 μs	0.77 s	3.26 s	4.20 s	632472 B	632472 B
pre demosaicing	0.32 ms	0.55 ms	0.55 ms	77.07 μs	78.03 μs	79.74 μs	0.77 s	3.26 s	4.20 s	36 B	632484 B
demosaicing_0	0.41 s	1.04 s	1.05 s	60.58 μs	61.54 μs	66.91 μs	0.77 s	3.26 s	4.20 s	157704 B	163370 B
demosaicing_1	0.53 s	1.05 s	1.07 s	60.30 μs	62.63 μs	66.91 μs	0.77 s	3.26 s	4.20 s	160296 B	163370 B
demosaicing_2	0.61 s	1.04 s	1.07 s	58.70 μs	60.98 μs	66.91 μs	0.77 s	3.26 s	4.20 s	160296 B	163370 B
demosaicing_3	0.60 s	1.05 s	1.05 s	58.86 μs	61.32 μs	66.91 μs	0.77 s	3.26 s	4.20 s	157704 B	163370 B
post demosaicing	0.70 s	2.28 s	2.40 s	149.09 μs	159.04 μs	179.65 μs	0.77 s	3.26 s	4.20 s	338964 B	2538900 B
post processing	0.77 s	3.26 s	4.20 s	73.63 μs	76.26 μs	80.43 μs	0.77 s	3.26 s	4.20 s	620750 B	1255992 B
write result	0.36 ms	0.53 ms	0.57 ms	37.02 μs	37.32 μs	56.37 μs	0.77 s	3.26 s	4.20 s	368 B	623460 B
process network		3.26 s	4.20 s		159.04 μs	179.65 μs		3.26 s	4.20 s	2176.4 KB	6188.3 KB

Table 7.2: Measured stabilization time $t_{stab,v}$ vs. its upper bound $t^*_{stab,v}$ for each process of the Demosaicing application. In addition, the maximum context size D_v is compared to its upper bound D^*_v. Notice that $t_{stab,v}$ and $t^*_{stab,v}$ are dominated by worst case values of $t_{phase1,v}$ and $t^*_{phase1,v}$ respectively. The correlation is purely incidental, and does not hold for all use cases, see Table 7.6.

process	**context size for** S_1
load image	48
pre processing	24
pre demosaicing	36
demosaicing_0	72
demosaicing_1	72
demosaicing_2	72
demosaicing_3	72
post demosaicing	84
post processing	152
write result	68

Table 7.3: Context size *exclusively* due to S_1.

time that a process is executing without calling a communication primitive; and the maximum amount of data that is transmitted per outgoing channel and iteration. The time of the longest compute segment was measured over each frame of an example video with resolution 320×240 pixels.

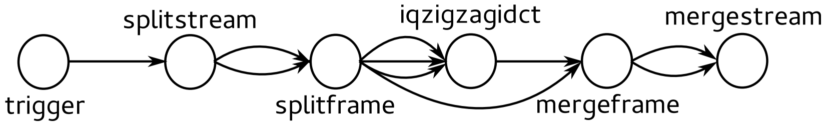


Figure 7.4: The MJPEG application.

Similar to the first benchmark example, we compare the measured stabilization time $t_{stab,v}$ of each process v with its upper bound $t^*_{stab,v}$, see Table 7.6. The experiment was repeated 20 times and the average and maximum results are reported in Table 7.6. The results confirm the trend observed with the Demosaicing application. In particular, all processes

process	**binding**	**max. execution time**	**port**	**max. data / iteration**
trigger	02	58 μs	0	4 B
splitstream	04	167 μs	0 1	4 B 10 KB
splitframe	06	98 μs	2 3 4 5	307.2 KB 64 B 4 B 8 B
iqzigzagidct	18	875 μs	3	76.8 KB
mergeframe	20	116 μs	2 3	7.68 KB 8 B
mergestream	22	114 μs		

Figure 7.5: Characteristics of the MJPEG decoder application. *Binding* refers to the core of the SCC on which the process executes.

process	$t_{phase1,v}$		$t^*_{phase1,v}$	$t_{phase2,v}$		$t^*_{phase2,v}$	$t_{stab,v}$		$t^*_{stab,v}$
	avg	**max**		**avg**	**max**		**avg**	**max**	
trigger	55.5 μs	129 μs	155 μs	20.1 μs	23.7 μs	24.1 μs	401 μs	671 μs	899 μs
splitstream	102 μs	157 μs	167 μs	31.1 μs	44.8 μs	47.7 μs	412 μs	698 μs	923 μs
splitframe	48.8 μs	96.6 μs	98.9 μs	49.4 μs	76.4 μs	77.5 μs	430 μs	699 μs	952 μs
iqzigzagidct	380 μs	653 μs	875 μs	47.5 μs	74.2 μs	78.4 μs	428 μs	716 μs	953 μs
mergeframe	71.1 μs	116 μs	116 μs	43.1 μs	65.1 μs	66.9 μs	424 μs	683 μs	942 μs
mergestream	53.8 μs	107 μs	114 μs	24.7 μs	37.2 μs	37.6 μs	405 μs	687 μs	913 μs
process network		653 μs	875 μs		76.4 μs	78.4 μs		699 μs	953 μs

Figure 7.6: Measured stabilization time $t_{stab,v}$ vs. its upper bound $t^*_{stab,v}$ for each process of the MJPEG decoder.

stabilized before the expected time bounds. In many cases, the bounds are actually very accurate. For the MJPEG decoder application, the upper bound is mainly composed of the maximum execution time $t^*_{c,v}$ of the iqzigzagidct process.

Time and Memory Overheads. The time and memory overhead generated by the additional code required to accomplish process network stabilization is presented in Fig. 7.7. The time overhead is mainly due to the additional logic to check for the `stop` token from the master and related housekeeping activities. In particular, an individual checking for the `stop` token has taken on average $43.01\,\mu$s.

application	**memory overhead**	**time overhead**
Demosaicing	8624 B	43.01 μs($< 0.05\%$)
MJPEG decoder	7104 B	43.01 μs($< 0.05\%$)

Figure 7.7: Overhead in terms of execution time and binary code size for adding the ability to stabilize compared to the original implementation.

7.9 Closing Remarks

With this chapter, this thesis has covered three important approaches to building a reliable multiprocessing system. This chapter proposed a new technique to recover from a fault, targeted to multiprocessing applications modeled after general data-flow process networks, such as Kahn Process Networks. Specifically, this chapter covers the challenges associated with stabilizing a general data-flow process network, which is a pre-requisite for a correct migration of the given process network. The proposed technique has been shown to be lightweight, and preserves the original functionality of the network. The correctness of the technique has been shown to be independent of the temporal characteristics of the system and the topology. We have shown that if the token communication time and *process compute time* are upper bounded, then an upper bound on the overall stabilization time can be calculated. Furthermore, this chapter provides simple mathematical expressions for estimating upper bounds on the maximum stabilization time and corresponding maximum context sizes for individual processes, and also the entire process network. These expressions can be used to make a decision as to whether it is feasible to attempt recovering from a fault in scenarios wherein the system does not fail immediately after experiencing a fault. We noted that owing to Intel SCC's high speed on-chip network, the maximum stabilization time is dominated by the maximum time a process can execute without calling any communication primitive, i.e., *process compute time*. The system designer may choose to reduce the stabilization time by inserting additional checks for events in the process' compute segments. Finally, detailed results from experiments on the Intel SCC baremetal platform and two realistic multiprocessing applications were presented, validating the ideas presented in this chapter.

8

Closing Remarks

8.1 Overall Summary

This thesis presented techniques useful for building a reliable system using three mutually orthogonal design pillars:

1. Start by analyzing the runtime environment in order to eliminate those use-cases which can cause (thermal) faults, i.e., fault avoidance,
2. Incorporate into the design, the capability to tolerate faults transparently to the observer, i.e., fault tolerance, and
3. Recover from faults transparently to the observer, i.e., fault recovery.

Though this thesis focuses on thermally induced faults (or simply, thermal faults), the ideas can be applied to other classes of faults as well. All concepts proposed in this thesis were tailored to time and resource constrained systems, and were validated on state-of-the-art systems.

Fault Avoidance

Avoiding thermal faults requires an accurate model of the system, specifically, a thermal model of the processor. The thermal model is then used to simulate offline, the temperature of the processor, given necessary information about the runtime environment, e.g., set of applications, corresponding time-schedule, and the mapping between applications and cores of the processor. This information is then used to decide apriori whether the system is in the danger of overheating at runtime, and hence effective for avoiding thermally induced faults. To this end, this thesis described a procedure to construct a thermal model for any processor which features at least one temperature sensor. Furthermore, we also described a calibration based method to account for the thermal effect of active cooling packages mounted on the processor (e.g, a system fan). The strength of the proposed approach lies in the minimal information necessary for constructing the thermal model of the given processor. All the necessary information required for realizing a thermal model is automatically extracted from the processor by a series of calibration experiments. Thus, the entire procedure for constructing the model can be fully automated, only requiring access to the physical processor for completing the calibration (and optionally, validation) steps. The thermal model construction approach proposed in this thesis is in sharp contrast to the state-of-the-art simulator based approaches such as Hotspot, which require extensive knowledge of the physical properties of the target system (e.g., floorplan of the processor, detailed power model, thermal properties of materials in order to obtain accurate simulation results.

We then proceeded to use the thermal model for estimating the worst case peak temperature of the processor, using abstract information about the system, such as a simplified thermal model, and arrival curves indicating the workload. Though the worst case peak temperature estimation is based on (significantly) simpler single pole thermal models, we showed that all conclusions apply equally when the thermal model used in complex.

Fault Tolerance

Fault tolerance was achieved using an adaptation of the well known N-version programming approach. The original N-version programming approach was modified to detect and tolerate faults requiring only two replicas of the application which execute in parallel to each other. We assume that the application to be made fault tolerant fails-silently, a requirement in many safety critical engineering domains, such as automotive electronics. This assumption also simplifies the problem from general purpose fault detection to detection of timing faults. The proposed approach is able to detect faults by simply observing the fill levels of selected FIFO buffers in the system, and thus avoids the use of any timer resources. This is in contrast to most other approaches for detecting faults in real time systems which depend on a series of timers, making such approaches difficult to scale with the size and timing complexity of applications.

Fault Recovery

Finally, we proposed a technique to recover from faults wherein the system does not immediately fail after having experienced a fault. Thus, the proposed approach is especially suited to recovering from thermal faults, wherein the system is said of have experienced a thermal fault if its temperature exceeds a pre-set threshold smaller than the critical temperature. The system continues to be available until its temperature has exceeded the critical temperature. This limited time can be used to migrate an application from a faulty location (e.g., an overheated core) to a new, fault-free location. The pre-requisite for correct migration is that the context of the application be stored correctly, and this in turn requires that a technique be devised which can guide application(s) to a *stable state* wherein the application context ceases to change any further. The proposed technique is specifically designed to be used with applications modeled after general dataflow process networks, and is therefore applicable to a large class of multiprocessing applications. Unlike most other approaches, the proposed solution does not require any special operating system or hardware.

8.2 Open Research Challenges

Improving Thermal Models

The thermal models proposed in this thesis are built using minimal information and assumptions necessary. For instance, it is assumed that the processor for which the thermal model is required features only one or more temperature sensors. However, modern embedded and mobile processors increasingly feature performance and energy counters, which may be used instead of utilization traces to generate more accurate models, see [SS13]. Furthermore, estimation of temperature traces is currently limited to those use cases wherein the set of applications to be executed on the processor have a corresponding thermal model. It may be possible to generalize the approach by constructing thermal model for a *class* of applications, rather than on a per application basis. Specifically, one may try to categorize applications based on how these affect the temperature of the processor. One approach may be to classify all compute intensive applications into one class, whereas all cache intensive applications belong to another class. Thermal models are then derived on a per-class basis. For estimating temperature traces, an application will need to be first classified into one of the available classes, and the corresponding thermal model used. Another possible approach is to classify individual thermal models, based on similarities in the impulse and step responses, and then choose (or derive) a single thermal model for each class.

Reducing Thermal Stress in the Processor

It is known that the reliability of the processor decreases if it experiences rapid changes in its temperature over time (i.e., temporal thermal gradients) In addition, for the case of a multicore processor, differences in the temperature across neighboring cores (i.e., spatial thermal gradients) are also known to reduce reliability, see [Cle03, BSBN14]. To this end, approaches have already been proposed which can statically determine an optimal computing strategy (e.g., mapping between applications and cores, corresponding schedules, etc), see [CRWG08]. However, for some use cases, such as the case of a server servicing a variety of requests from users, it may not be feasible to determine any solution offline. This scenario requires the development of an online control engine which leverages the speed and accuracy of the thermal models, in order to continuously refine the computing strategy. To this end, the control engine may utilize the following *knobs*: migrate applications, change processor clock speeds, modify time schedules, and even adjust cooling as and when required. An early investigation into the topic revealed that whereas simple solutions based on heuristics can be designed, a search for an optimal solution is still open. As an example, we have attempted the problem by using the concept of per-core *thermal bins* as follows: First, we co-schedule a subset of applications with similar steady-state temperatures onto a single core, provided none of the applications miss their deadlines. Furthermore, any time-slack wherein the core is idle is removed by scaling down the processor clock speed appropriately. Thus, the chosen strategy minimizes temporal thermal cycles, see [Mü15]. Reduction in spatial thermal cycles can be ensured by placing thermal bins with similar mean temperatures in vicinity. This approach presents promising results, but as indicated before, a search for an optimal solution is still open.

Computer Security

Recent research trends in the area of computer security indicate a growing interest in understanding temperature based covert side channels, see [BDK$^+$09]. The underlying idea is to observe the changes in temperature of a core as it executes applications in order to derive useful information, such as messages. Conventionally, reducing information leakage (or exfiltration) has been achieved by partitioning computer resources. That is, each security critical application executes on a dedicated core, accesses dedicated memory, and may even use dedicated communication resources. However, these techniques do not work when the covert channel used for leaking information is based on modulating the temperature of (a core of) a processor, since all the computation and communication units are fabricated on the same silicon die. We have attempted to explore the rate at which two colluding applications (e.g., a sender and a receiver), isolated from each other using resource partitioning approaches, are able to communicate to each other, see [MRR$^+$15] The communication rate depends on several factors, such as the background noise in the system, the physical distance separating the two colluding parties, cooling, and also how the colluding applications are designed. The communication rate increases when the sender executes compute intensive instructions, spiking the temperature of its core rapidly. These temperature changes are sensed by the receiver, and converted to useful information. Furthermore, due to high thermal resistivity of silicon, the communication rate increases if the colluding parties are physically close, with the optimal being that the colluding parties execute on the same core, of course, partitioned in time. Our approach has thus far been exploratory in nature, trying out different combinations of applications, separation distances, processor clock speeds and the like. However, derivation of an upper bound on the maximum possible communication rate, as a function of above discussed factors will be a useful contribution to the scientific community. The thermal models derived in this thesis may be a good starting point for such an endeavor as these models already contain expressions to estimate the temperature of the processor accounting for the factors discussed above.

A

Real Time Computing on the Intel SCC

Summary

This chapter describes the experience in using the Intel Single Chip Cloud Computer (SCC) for real time applications. Chapters 6 and 7 of this thesis use the concepts described in this chapter. At the time the author worked on this thesis, Intel SCC was the only many-core platform capable of supporting real time computation, and also had an outstanding community support. Specifically, this chapter proposes design strategies which enable high performance multiprocessing applications to achieve predictable communication latencies, computation times and *reaction latencies* over on-chip many-core platforms, such as the Intel SCC. The ideas presented in this chapter can be easily applied to other similar platforms, notably the Kalray MPPA-256 processor, see [de 13].

A.1 Introduction

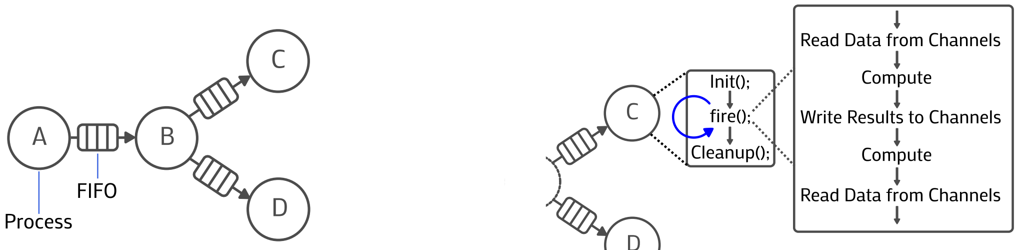
Most multi- and many-core systems available today are primarily designed to support high performance computing, rather than real time performance. Common examples of such systems are the Intel Xeon Phi co-processor, the Intel SCC co-processor, the Parallella co-processor, and the P2012 co-processor see [JR13, MRL$^+$10, VEMR14, BFFM12]. However, some co-processors, such as the Intel SCC, and the more recent Kalray MPPA-256 processor also support low latency and low jitter performance, which can be used for implementing real time applications. Specifically for the case of the Intel SCC, the reduction in latency and jitter comes from simpler P54C architecture in the cores combined with a high bandwidth and deterministic on-chip communication network. Recent trends indicate that processors will continue to feature increasing number of cores, since such processors allow the developers implement truly parallel applications. In addition, the availability of multiple cores in a processor offers an opportunity to optimize power costs by shutting down cores which are not used. This chapter presents design ideas which can be used to implement applications that require predictable performance characteristics when executing on state-of-the-art many-core (co-) processors. Succinctly, an application is said to have predictable performance characteristics if the time required by the application to react to events, compute, and communicate can be statically estimated and bounded, see Section A.4. Moreover, it may be required that a given multiprocessing application react to events (e.g., control messages) in a bounded time. With this motivation, this chapter proposes software design architecture with the following objectives:

1. Enable an application to execute with a predictable timing performance, and
2. Enable an application to react to control messages in a predictable time-frame (henceforth, *bounded reaction latency*).

The proposed approach does not assume any special operating system features or services, and is therefore portable across platforms. We assume a general data-flow process network model of computation, which is extensively used to design and implement streaming applications (e.g., medical ultrasound). All proposed design concepts have been validated on the Intel SCC operating in the baremetal configuration (i.e., no operating system is used). It must be emphasized that merely using the SCC in the baremetal configuration does not *automatically* lead to predictable timing performance.

A.2 Related Work

Although it is possible to port one of several multiprocessor real-time operating systems (RTOS) to the Intel SCC, an RTOS may consume a significant portion of the available computing resources on the chip, thus motivating the need for a baremetal (i.e., without any operating system on the processor) solution, see [Åk02]. As a result, C-language based programming frameworks such as BareMetalC and BareMichael are available which allow the applications to run directly on the SCC cores, without the need of an operating system, see [MRL$^+$10, Z$^+$12]. In addition, baremetal-compatible communication libraries such as RCCE, iRCCE provide intra- and inter core communication, see [vdWMH11, C$^+$11]. Puffitsch *et. al.* use the baremetal approach to execute task-sets in real time on the SCC,



(a) A Data flow Process Network.

(b) Structure of a Process.

Figure A.1: Structure of the process network (left), and a process (right).

but their approach is restricted to periodic tasks with no pre-emption, and tasks are not required to respond to events within a bounded latency, see [PNP13]. In contrast, this chapter assumes general data-flow process networks, which must react to events within a statically bounded time.

A.3 Background

A.3.1 The Intel SCC Processor

The SCC processor is a 48-core experimental processor from Intel, featuring 24 organized into a 4×6 grid and linked by a 2D mesh on-chip network. A tile contains a pair of P54C processor cores, a router, and a $16\,\mathrm{KB}$ block of SRAM, which is used for message passing. Each core has L1 instruction and data caches (16KB each) and a unified level 2 (L2) cache (256KB), see [MRL$^+$10] . Also see Section 6.6 for further details.

A.3.2 Data Flow Process Network Model of Computation

We assume a set of concurrently executing deterministic processes, communicating with each other in a *point-to-point* manner, via bounded FIFO buffers. A process attempting to read (write) an empty (full) FIFO buffer blocks (or stalls) till the read (write) operation can be successfully completed. A process can only execute (*fire*) when it has acquired sufficient data to proceed with its computation. Process networks have natural support for parallelism making them an attractive model for describing multiprocessing systems, specially digital signal processing systems.

A.3.2.1 Structure of the Process

The functionality of each process in a network is split into three parts: *initialization*, *execution* and *cleanup*, with each part executed by a call to a pre-defined function, see Figure A.1. The initialization function (`init`) prepares a process for execution, by initializing variables, and allocating memory, if necessary. The main functionality of the process is encapsulated in the `fire` function, and runs as long as necessary, continuously reading data from input channels, computing, and writing results to one or more output channels. The `fire` function therefore, consists of *compute* segments, surrounded by channel read and/or channel write segments. The compute segments involve purely processing of already acquired data, and the time to execute each of these segments depends on the nature of acquired data. The segment with the longest computing time is called the *dominating segment*. The `cleanup` function is called once the process terminates its core functionality, and is used for returning resources to the system, or gathering performance statistics.

A.3.2.2 The *Master* Process and the Modified Process Structure

Controlling the behavior of the application at runtime is achieved via a special process, called the *master process*, which can send appropriate control messages to the application at runtime. For brevity, we only discuss the `stop` control message, which is used to indicate to a process that it must exit its `fire` function and proceed to execute the `cleanup` function. Its structure follows the layout of the process in Figure A.1b.

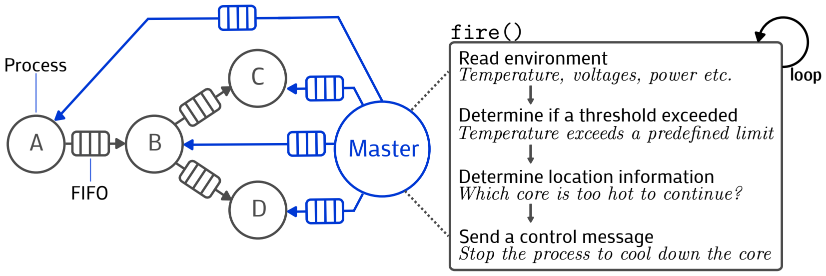


Figure A.2: An example of the *master process*. Additional control channels are also shown.

Further, in order that a process can react to control messages, its structure has been modified as shown in Figure A.3. After every call to `Read` (or `Write`) function, the process checks for the control message (for example, `stop`), and takes appropriate action. Inserting a check for control message after every call to `Read` or `Write` is automatically done at the precompilation stage. In the rest of this chapter, the process from the original network is denoted simply as a *process* in order to distinguish it from the *master process*.

A.4 Achieving Predictable Timing Characteristics

A.4.1 Definitions

For the purpose of this chapter, *predictable timing characteristics* consists of three parts: (i) observed time to execute any given compute segment

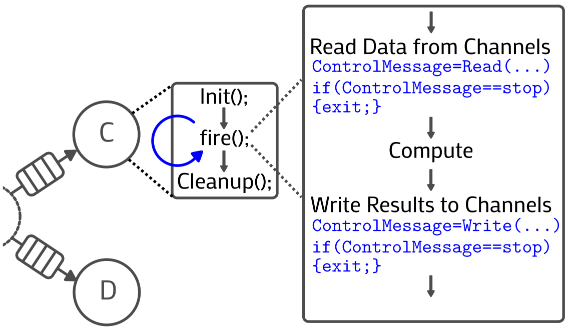


Figure A.3: Modified process structure.

with an application is within a known bound, (ii) observed time for an application to successfully transmit a fixed size message is within a known bound, and (iii) observed time for an application to react to a control message from the master is within a known bound.

We define reaction time of a process as:

$$R_p = t_{p,C^*} - t_{m,C^*} \tag{A.1}$$

where R_p is the reaction time of a process p, t_{p,C^*} is the time when a process receives the control message, and t_{m,C^*} is the time when the master queues a control message for transmission (i.e., when a *non-blocking* call to transmit the message from the master to the process is issued)

A.4.2 Proposed Solution

The proposed solution has been divided into the following sections:

1. A new communication interface such that for any process, the delay in receiving a control message is independent of the size of data messages being exchanged between processes, thus ensuring bounded reaction times;

2. Mapping between processes and cores which reduces uncertainties in communication times between processes;

3. Eliminating factors that may lead to unpredictable compute segment times.

A.4.3 A New Communication Interface

We do not place any limit on the length of data messages exchanged between processes. However, arbitrarily long messages lead to arbitrarily long communication times, leading to unpredictable delays in reading a control message from the master. Clearly, in order to ensure bounded reaction times, the control message communication must be decoupled from normal data communication. In addition, as per the assumed model of computation, it must be ensured that a process attempting to read a channel with insufficient data must not proceed further as long as all data has not been acquired. Therefore, we propose a `Read`[1] function, which is called by the process to acquire data with the application programming interface (API) as follows:

```
ControlMessage = Read (ID, *buffer, size, source);
```

The function returns `ControlMessage` holding the value of any control message, if received. Here, `ID` is the identifier of the request, useful when requests may be queued, `*buffer` is the memory location where the received message will be saved, `size` is the length in bytes of the message to expect, and `source` is the identifier of the core from which to receive the message. The structure of the `Read` API closely resembles standard developer APIs found in commonly used communication libraries, such as the RCCE, or the iRCCE, see [C$^+$11, CLRB11]. In order to decouple data messages from control messages, the `Read` function:

1. Receives an arbitrary sized message in chunks not exceeding a fixed size;
2. Intersperses checking for a control message from the master with reading data messages;

As indicated earlier, if a process receives a control message (e.g. `stop`), the process exits its `fire` function and proceeds to `cleanup`, see Figure A.3.

In order to ensure that the MPB has guaranteed space for holding control messages, we require that:

$$\texttt{CHUNK\_SIZE} + |C^*| \leq |MPB| \tag{A.2}$$

Where $|C^*|$ is the maximum size of the control message, `CHUNK_SIZE` is the maximum *chunk size* of the data message, and $|MPB|$ is the size of the MPB on each tile of the Intel SCC. Very small values of chunk size results in a large number of calls to the communication library, leading to inefficiency, but also decrease the latency with which a process checks for a control message. Thus, the choice of chunk size reflects a trade-off made by the designer at design time. It is clear that the sender process must cooperate with the receiver process for this technique to work. Specifically, the sender splits an arbitrarily large messages into chunks not exceeding `CHUNK_SIZE`, with chunks being transmitted (and received) in the correct order. The receiver must then assemble chunks in the correct order to receive the complete message. The above discussion is summarized in Figure A.4. Referring to Figure A.4, the receiver process expects `n` bytes from the sender, and therefore calls its `Read` function. The communication interface reads the message in chunks, interleaving with reading (if

---

[1] The discussion for the corresponding `Write` function is analogous.

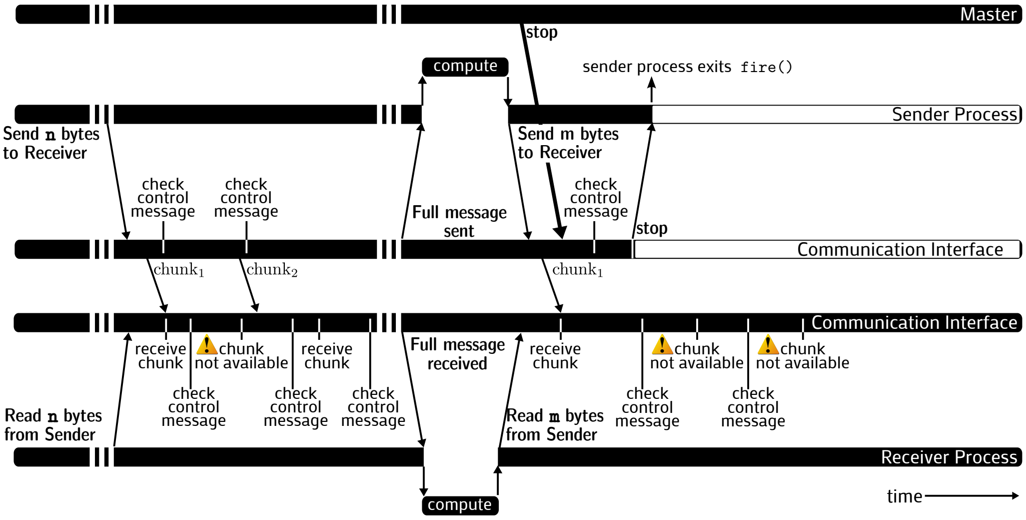


Figure A.4: Timeline of two communicating processes along with a master.

any) control message from the master. Notice that all processes (including the master) are asynchronous, and there may be instants when the receiver expects to receive a data chunk from the sender, but is not successful. In such situations, the communication interface must not block, and must proceed to checking for the control messages from the master. In order to ensure that the communication interface does not return to the caller without having acquired sufficient data, the communication interface keeps a running count of total message size received with each new chunk. Such non-blocking receive for data messages justifies the use of non-blocking communication library, such as the iRCCE.

Timing Bounds

The worst case reaction time of a process is defined as follows:

$$R_p^* = t^*_{read|write} + S_p^* + t^*_{m,C^*} \tag{A.3}$$

where R_p^* is the worst case reaction time of process p, $t^*_{read|write}$ is the maximum time to completely receive (or send) one `CHUNK_SIZE` of message, S_p^* is the length of dominating computation segment of process p, and t^*_{m,C^*} is the maximum time to transport the control message from the master to process p.

A.4.4 Process-to-Core Mapping

The mapping of processes to cores impacts timing properties of an application due to at least two factors:

Process Interference

Unless carefully scheduled, two or more processes sharing the same core result in non-deterministic demands on the available computing resources (such as MPB, tile router), thereby causing non-real time behavior. The timing performance of the application is improved by mapping only one process to a tile (and switching off the unused core).

Communication Traffic Interference

If message streams between more than one pair of sender-receiver processes contend at an intermediate router, then, timing disturbance due to such cross traffic may be observed at both pairs of processes. Therefore, processes must be mapped in a manner that minimizes cross traffic at all active routers on the chip, see [ZM12].

A.4.5 Eliminating other interferences

Timing predictability of an application is further improved by disabling all interrupts and L2-caches, which is a common practice in the design of real time systems.

A.5 Experiments and Results[2]

A.5.1 Methodology

Since reaction time (see (A.3)) is influenced by dominating segments as well as communication, the overall objective of this section is to show that R^* computed using (A.3) upper bounds all values of reaction times observed when running real applications. In order to determine representative values for $t^*_{read|write}$, S^*_p and t^*_{m,C^*}, a series of calibration experiments are performed on the Intel SCC:

A.5.1.1 Message Read-Write Timings

A simple producer process was mapped to core 0 and a simple consumer process was mapped at hop distances of 1,2,3 and 4 from the producer. The message size transmitted by the producer ranged from 32 bytes to 3KB. Timings were recorded under ideal and noisy conditions, and were measured with reference to the core's local time-stamp counter (TSC). In order to ensure that the reported timings are correct, the sender and receiver clocks were first synchronized using a barrier, and then, the sender transmitted its local time stamp to the receiver. The receiver computed the message travel time by referencing the received time-stamp with its own on-core time stamp counter, and thereby avoiding costly accesses to

---

[2] Experiments were performed on a remote SCC operating in the baremetal configuration.

the global time-stamp counter. In the first set of experiments, only the producer-consumer pair was active on the chip, and the observed timing model is summarized in (A.4). The reported model matches well with the timings reported in [Mat13].

$$t_{p \to p'}(\text{clock cycles}) = 5.289|P| + 57.53|H| + 894.4 \tag{A.4}$$

Where $t^*_{p \to p'}$ is the worst-case time in delivery a given message, $|P|$ is the packet size of the message in bytes, and $|H|$ is the *hop-distance* between the sender and the receiver (measured along the message transport route). Next, one traffic generator process was mapped onto one core in each tile of the SCC, with the objective of creating heavy traffic at each router of the SCC. The producer and consumer processes did not shared any core with the traffic generators. The corresponding worst case time is presented in (A.5).

$$t^*_{p \to p'}(\text{clock cycles}) = 4.908|P| + 9921 \tag{A.5}$$

Notice that the worst case time is independent of the hop distance between cores. Each model was derived out of a total of 800 observations. Note that the reported times include *both* the message transmission time, *and* the time to read the full message by the consumer.

A.5.1.2 Dominating Segments

Dominating segments were evaluated by carefully isolating computing segments from communicating segments in each process of the selected application. Next, the application was run with an input known to lead to high compute times. For signal processing application, these inputs are known in advance. For instance, for Demosaicing, an input image with lot of high-frequency components will lead to high compute times. From the calibration runs, the dominating segment of each process is calculated. Once all calibration runs are complete, the upper bound on reaction times (R^*) is calculated using (A.3).

A.5.2 Results

For experiments, two large applications, the *Demosaicing* [Li05b], and the *motion-JPEG* (MJPEG) decoder are selected. The SCC was booted with the following parameters: tile clock: 533MHz, router: 800MHz, DDR3: 800MHz, with L2-caches disabled. The size of control messages was set to 32 bytes, and `CHUNK_SIZE` was set to 3KB. Both applications are compute and communication intensive.

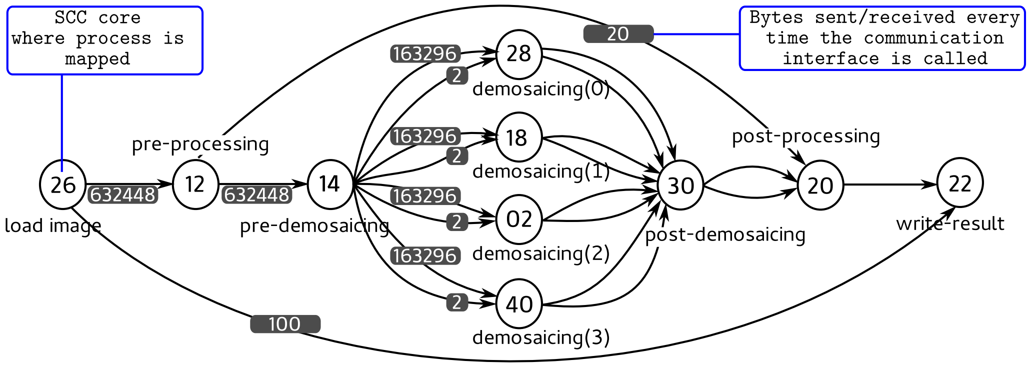


Figure A.5: The Demosaicing application with the mapping details. Size of data messages exchanged between processes is also shown. Master process was mapped to core 24.

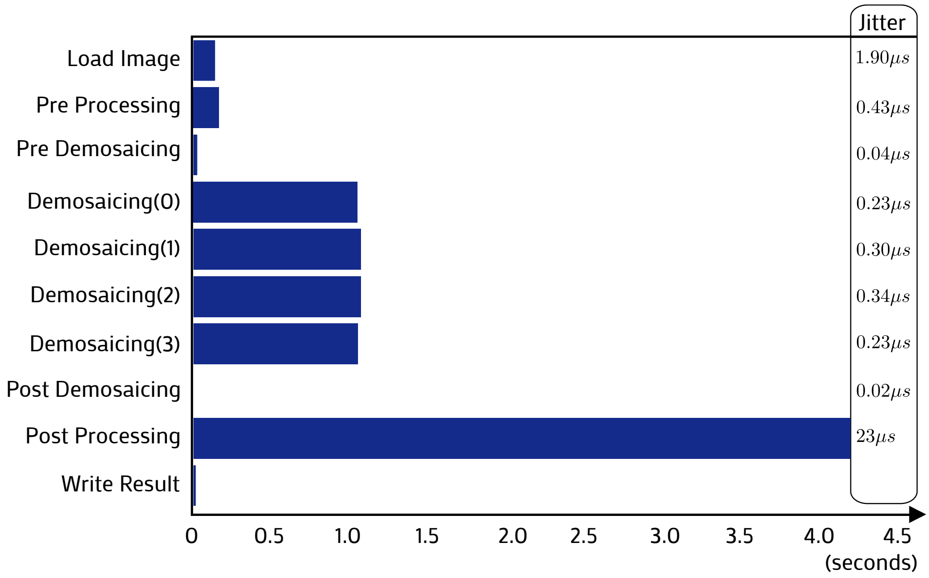


Figure A.6: Lengths of dominating segments for each process in the Demosaicing application.

Reaction Times for the Demosaicing Application.

The Demosaicing process network is shown in Figure A.5. Notice that the process-to-core mapping ensures that one tile does not host more than one process, and cross traffic at routers is minimized. Reaction times for each process in the Demosaicing application are shown in Figure A.7, with lines representing observed values from experiments, while the bars represent the bounds computed during calibration runs. Notice that the bounds are tight.

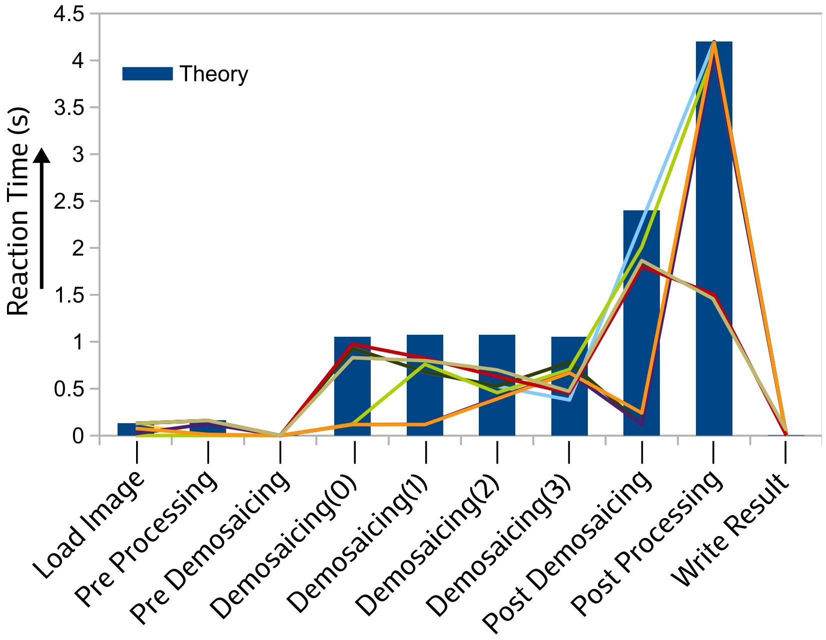


Figure A.7: Reaction Times for the Demosaicing application for 10 observations. Lines show observed timings during experiments.

Reaction Times for the Motion-JPEG Decoder.

The MJPEG decoder process network is shown in Figure A.8. The splitstream and splitframe processes search for certain markers in the encoded image. The amount of time spent in searching for the marker depends on the frame itself, and hence, the dominating compute segments in these processes show large variations. Consequently, for calculating worst case reaction time R_p^*, the maximum size of corresponding dominating segments is used.

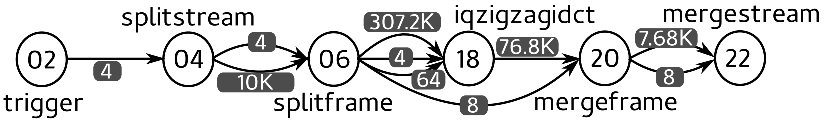


Figure A.8: The Motion JPEG decoder. Master process was mapped to core 16.

Figure A.9 shows the lengths of dominating segments for the MJPEG application, while Figure A.10 compares the predicted value of reaction times to those observed over 10 experiments involving 30 frames each. Notice that unlike the Demosaicing application, reaction times in MJPEG application are not determined solely by the dominating segments, but are also influenced by communication times. Also notice that the *iqzigzagidct* process communicates messages of size 76.8KB, which according to (A.5) would take $725\mu s$. Unless the entire message was communicated in

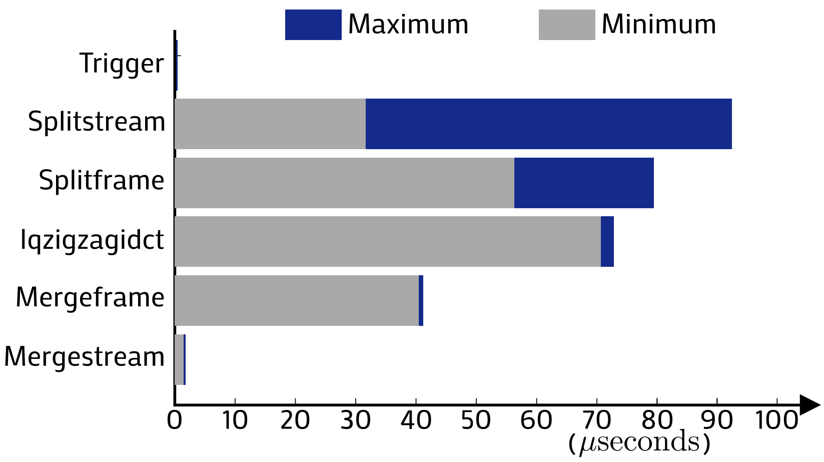


Figure A.9: Size of dominating segments for the MJPEG decoder.

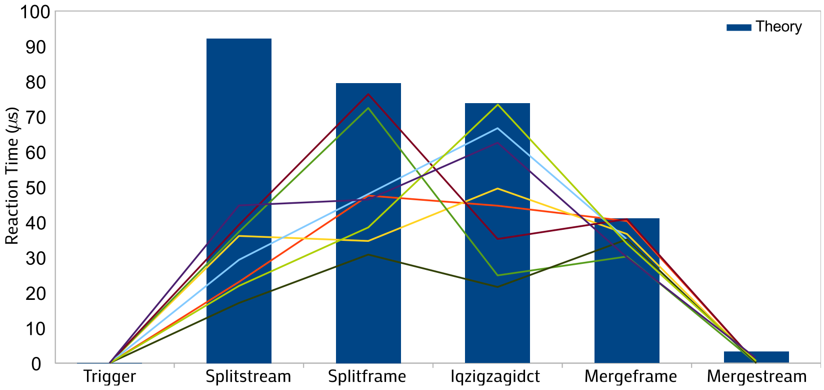


Figure A.10: Reaction Times for the MJPEG decoder for 10 observations.

chunks, the reaction times would have been an order of magnitude more than what is reported in Figure A.10.

A.5.3 Overhead

The definition of the new communication interface was common to both applications, and the additional logic for chunk-based message communication resulted in a mere 528 byte overhead. The chunk-based message communication *improved* the timing efficiency of message communication, leading to a 4.7% improvement in per-frame timing for MJPEG decoder. This is because chunk-based communication mitigated the number of times shared memory was accessed during message passing, which more than compensated for larger number rounds required to send the complete message. Any time overhead for the Demosaicing application was negligible since timings in this case are determined by dominating compute segments.

A.6 Summary

This chapter demonstrated that it is indeed possible to achieve predictable runtime performance, and bounded reaction time of applications on the Intel SCC processor. To this end, a non-blocking communication strategy was proposed which interleaves data and control communication, and also complies with the execution semantics of general data flow process networks. This strategy enables a process (or an entire process network) to react to events with a bounded and predictable latency. Furthermore, the above-mentioned strategy was encapsulated in an RCCE- (or MPI-) like communication API, making it possible to easily incorporate the proposed strategy into existing multi- and many-core application design frameworks. Experiments involving two representative applications validate the proposed technique and show that predictable performance, including bounded reaction latencies can be achieved easily on baremetal SCC, and indeed on any many-core processor. Last, the techniques applied in the chapter for achieving predictable runtime performance are generic, and can therefore be applied to other many-core processors, such as the Kalray MPPA-256, or the Parallella processor.

Bibliography

[A+08] Andrea Acquaviva et al. Assessing Task Migration Impact on Embedded Soft Real-Time Streaming Multimedia Applications. *EURASIP J. Embedded Syst.*, pages 9:1–9:15, 2008.

[A+09] Gabriel Marchesan Almeida et al. An Adaptive Message Passing MPSoC Framework. *Int'l J. of Reconfigurable Computing*, 2009.

[And13] Andreas Olofsson. tinyurl.com/kre2k5z, 2013.

[Art08] Artist. Benchmarks. `http://www.artist-embedded.org/artist/Benchmarks.html`, 2008.

[B+06] Stefano Bertozzi et al. Supporting Task Migration in Multi-Processor Systems-on-Chip: A Feasibility Study. In *Proc. DATE*, pages 15–20, 2006.

[BBB+11] P. Bourgos, A. Basu, M. Bozga, S. Bensalem, J. Sifakis, and K. Huang. Rigorous system level modeling and analysis of mixed hw/sw systems. In *Formal Methods and Models for Codesign (MEMOCODE), 2011 9th IEEE/ACM International Conference on*, pages 11 –20, july 2011.

[BDK+09] Julien Brouchier, Nora Dabbous, Tom Kean, Carol Marsh, and David Naccache. Thermocommunication, 2009. david.naccache@ens.fr 14242 received 29 Dec 2008.

[Ber06] Bertoluzzo, M. and Buja, G. and Pimentel, J. Design of a Safety-Critical Drive-By-Wire System using FlexCAN. In *SAE Technical Paper 2006-01-1026*, SAE World Congress, Detroit, MI, USA, 2006.

[BFFM12] L. Benini, E. Flamand, D. Fuin, and D. Melpignano. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 983–987, March 2012.

[Bie11] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[BM01] D. Brooks and M. Martonosi. Dynamic thermal management for high-performance microprocessors. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pages 171–182, 2001.

[BP05] Nikhil Bansal and Kirk Pruhs. Speed scaling to manage temperature. In *STACS*, 2005.

[BSBN14] M. Becker, K. Sandstrom, M. Behnam, and T. Nolte. Limiting temperature gradients on many-cores by adaptive reallocation of real-time workloads. In *Emerging Technology and Factory Automation (ETFA), 2014 IEEE*, pages 1–8, Sept 2014.

[BTM00] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA ’00, pages 83–94, New York, NY, USA, 2000. ACM.

[C^+06] Sayantan Chakravorty et al. Proactive Fault Tolerance in MPI Applications via Task Migration. In *Proc. HPC*, pages 485–496, 2006.

[C^+11] C. Clauss et al. iRCCE: A Non-Blocking Communication Extension to the RCCE Communication Library for the Intel Single-chip Cloud Computer. Technical report, RWTH Aachen, 2011.

[C^+12] Emanuele Cannella et al. Adaptivity Support for MPSoCs Based on Process Migration in Polyhedral Process Networks. *VLSI Design*, pages 2:2–2:17, 2012.

[Cha85] K. Mani Chandy. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. on Computer Systems*, 3:63–75, 1985.

[CHK07] Jian-Jia Chen, Chia-Mei Hung, and Tei-Wei Kuo. On the minimization of the instantaneous temperature for periodic real-time tasks. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2007.

[Cle03] Clemens J.M. Lasance. Thermally driven reliability issues in microelectronic systems: status-quo and challenges. *Microelectronics Reliability*, 43(12):1969 – 1974, 2003.

[CLRB11] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl. Evaluation and improvements of programming models for the intel scc many-core processor. In *High Performance Computing and Simulation (HPCS), 2011.*, pages 525–532, 2011.

[CLS+06] S. Chakraborty, Yanhong Liu, N. Stoimenov, L. Thiele, and E. Wandeler. Interface-based rate analysis of embedded systems. In *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*, pages 25–34, 2006.

[CM10] Jin Cui and D.L. Maskell. High level event driven thermal estimation for thermal aware task allocation and scheduling. In *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*, pages 793 –798, jan. 2010.

[CRWG08] A.K. Coskun, T.S. Rosing, K.A. Whisnant, and K.C. Gross. Static and dynamic temperature-aware scheduling for multiprocessor socs. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(9):1127–1140, Sept 2008.

[CS06] Sung Woo Chung and K. Skadron. Using On-Chip Event Counters For High-Resolution, Real-Time Temperature Measurement. In *Thermal and Thermomechanical Phenomena in Electronics Systems, ITHERM*, pages 114–120, 2006.

[de 13] de Dinechin, B.D. and Ayrignac, R. and Beaucamps, P.-E. and Couvert, P. and Ganne, B. and de Massas, P.G. and Jacquet, F. and Jones, S. and Chaisemartin, N.M. and Riss, F. and Strudel, T. A clustered manycore processor architecture for embedded and accelerated applications. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pages 1–6, Sept 2013.

[DJL+04] Simon Derr, P Jackson, C Lameter, P Menage, and H Seto. Cpusets, 2004.

[DL80] L. Dugard and I. D. Landau. Recursive Output Error Identification Algorithms Theory and Evaluation. *Automatica*, 16(5):443–462, 1980.

[DSVA06] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems. In *Proc. SenSys*, pages 29–42, 2006.

[DZ11] Mian Dong and Lin Zhong. Self-constructive high-rate system energy modeling for battery-powered mobile systems. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 335–348, New York, NY, USA, 2011. ACM.

[EAH12] Thomas Ebi, Hussam Amrouch, and Jörg Henkel. Cool: Control-based optimization of load-balancing for thermal behavior. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '12, pages 255–264, New York, NY, USA, 2012. ACM.

[EBSA$^+$11] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[FCWT09] Nathan Fisher, Jian-Jia Chen, Shengquan Wang, and Lothar Thiele. Thermal-aware global real-time scheduling on multicore systems. In *RTAS*, 2009.

[FKC$^+$10] Y. Fu, N. Kottenstette, Y. Chen, C. Lu, X. Koutsoukos, and H. Wang. Feedback thermal control for real-time systems. In *RTAS*, 2010.

[FWP09] Xing Fu, Xiaorui Wang, and Eric Puster. Dynamic thermal and timeliness guarantees for distributed real-time embedded systems. In *RTCSA*, pages 403–412, 2009.

[GB10] Marc Geilen and Twan Basten. Kahn Process Networks and a Reactive Extension. In *Handbook of Signal Processing Systems*, pages 967–1006. Springer, 2010.

[GD05] D. Gerling and G. Dajaku. Novel lumped-parameter thermal model for electrical systems. In *Power Electronics and Applications, 2005 European Conference on*, pages 10 pp.–P.10, 2005.

[GDJ12] Meng Guo, D.V. Dimarogonas, and K.H. Johansson. Distributed real-time fault detection and isolation for cooperative multi-agent systems. In *American Control Conference (ACC), 2012*, pages 5270–5275, 2012.

[GRE$^+$01a] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization.*, WWC '01, pages 3–14, 2001.

[GRE$^+$01b] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Workload*

Characterization, 2001. WWC-4. 2001 IEEE International Workshop on, pages 3–14, Dec 2001.

[H$^+$09] Wolfgang Haid et al. Efficient Execution of Kahn Process Networks on Multi-Processor Systems Using Protothreads and Windowed FIFOs. In *Proc. ESTIMedia*, pages 35–44, 2009.

[H$^+$10a] J. Howard et al. A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. In *Proc. ISSCC*, pages 108–109, 2010.

[H$^+$10b] J. Howard et al. A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. In *Proc. ISSCC*, pages 108–109, 2010.

[HBZ$^+$14] J. Henkel, L. Bauer, Hongyan Zhang, S. Rehman, and M. Shafique. Multi-layer dependability: From microarchitecture to application level. In *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*, pages 1–6, June 2014.

[HCBK12] Kai Huang, Gang Chen, C. Buckl, and A. Knoll. Conforming the runtime inputs for hard real-time embedded systems. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 430–436, June 2012.

[HFFA11] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward Dark Silicon in Servers. *Micro, IEEE*, 31(4):6–15, July 2011.

[HG08] Liuping Wang Hugues Garnier, editor. *Identification of Continuous-time Models from Sampled Data*, volume XXVI of *Advances in Industrial Control*. Springer, 2008.

[HNPT13] J. Henkel, V. Narayanan, S. Parameswaran, and J. Teich. Run-time adaption for highly-complex multi-core systems. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2013 International Conference on*, pages 1–8, Sept 2013.

[HoMAECE07] Y. Han and University of Massachusetts Amherst. Electrical & Computer Engineering. *Temperature Aware Techniques for Design, Simulation and Measurement in Microprocessors*. University of Massachusetts Amherst, 2007.

[HSG$^+$09] Wei Huang, Kevin Skadron, Sudhanva Gurumurthi, Robert J. Ribando, and Mircea R. Stan. Differentiating the roles of IR measurement and simulation for power and temperature-aware design. In *ISPASS*, pages 1–10, 2009.

[HSL78] Jr. Hopkins, A.L., III Smith, T.B., and J.H. Lala. A highly reliable fault-tolerant multiprocess for aircraft. *Proc. IEEE*, pages 1221–1239, 1978.

[HSS96] A. Hlawiczka, J.G.S. Silva, and L. Simoncini. *Dependable Computing - EDCC-2: Second European Dependable Computing Conference*. Lecture Notes in Artificial Intelligence. Springer, 1996.

[HSS$^+$04] Wei Huang, M.R. Stan, K. Skadron, K. Sankaranarayanan, S. Ghosh, and S. Velusamy. Compact thermal modeling for temperature-aware design. In *Design Automation Conference, 2004. Proceedings. 41st*, pages 878 –883, july 2004.

[Jar14] Jarem Archer. tinyurl.com/lsp456h, 2014.

[JR13] James Jeffers and James Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

[Kah74] Gilles Kahn. The Semantics of a Simple Language for Parallel Programming. In *Proc. IFIP Congress*, pages 471–475, 1974.

[Kat13] Georgios Kathareios. Towards Exploiting Intra-Application Dynamism using an H.264 Codec. Master's thesis, ETH Zurich, Zurich, Switzerland, 2013.

[KB03] Hermann Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, Jan 2003.

[KBSM06] E. Karl, D. Blaauw, D. Sylvester, and T. Mudge. Reliability modeling and management in dynamic microprocessor-based systems. In *Design Automation Conference, 2006 43rd ACM/IEEE*, pages 1057 –1060, 0-0 2006.

[KSS$^+$12] Emre Kultursay, Karthik Swaminathan, Vinay Saripalli, Vijaykrishnan Narayanan, Mahmut T. Kandemir, and Suman Datta. Performance Enhancement Under Power Constraints Using Heterogeneous CMOS-TFET Multicores. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '12, pages 245–254, New York, NY, USA, 2012. ACM.

[KT11] P. Kumar and L. Thiele. Cool shapers: Shaping real-time tasks for improved thermal guarantees. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 468–473, June 2011.

[LBT01a] Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus — A Theory of Deterministic Queuing Systems for the Internet*, volume 2050 of *Lecture Notes in Computer Science*. Springer Verlag, 2001.

[LBT01b] Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer-Verlag, Berlin, Heidelberg, 2001.

[LDSY07] Yongpan Liu, Robert P. Dick, Li Shang, and Huazhong Yang. Accurate temperature-dependent integrated circuit leakage power estimation is easy. In *DATE*, pages 1526–1531, San Jose, CA, USA, 2007. EDA Consortium.

[LHL05] Weiping Liao, Lei He, and Kevin Lepak. Temperature and supply voltage aware performance and power modeling at microarchitecture level. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume Volume 24, Issue 7, pages 1042 – 1053. ACM, July 2005.

[Li05a] Xin Li. Demosaicing by successive approximation. *Trans. Img. Proc.*, 14(3):370–379, 2005.

[Li05b] Xin Li. Demosaicing by successive approximation. *Image Processing, IEEE Transactions on*, 14(3):370–379, March 2005.

[LKP$^+$10] Chanhee Lee, Hokeun Kim, Hae-Woo Park, Sungchan Kim, Hyunok Oh, and Soonhoi Ha. A Task Remapping Technique for Reliable Multi-Core Embedded Systems. In *Proc. CODES/ISSS*, pages 307–316, 2010.

[LL94] Chin Lu and Sau-Ming Lau. A Performance Study on Load Balancing Algorithms with Task Migration. In *Proc. TENCON*, pages 357–364, 1994.

[LPMS97] Chunho Lee, M. Potkonjak, and W.H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *EEE/ACM International Symposium on Microarchitecture.*, pages 330–335, Dec 1997.

[LTT08] Duo Li, Sheldon X.-D. Tan, and Murli Tirumala. Architecture-level thermal behavioral characterization for multi-core microprocessors. In *Proceedings of the 2008 Asia and South Pacific Design Automation Conference*, ASP-DAC '08, pages 456–461, Los Alamitos, CA, USA, 2008. IEEE Computer Society Press.

[Mar14] Martin Reisslein et. al., 2014.

[Mat13] Mattson, T.G. Using Intel's Single-Chip Cloud Computer (SCC). http://communities.intel.com/docs/DOC-19269, 2013.

[MC02] David Finkel Mark Claypool. Transparent Process Migration for Distributed Applications in a Beowulf Cluster. In *Proc. INC*, pages 459–466, 2002.

[Mic12] Michal Mienik, 2012.

[Mil98] Blair D. Milburn. Apparatus and method for initializing a master/checker fault detecting microprocessor, 1998.

[MMA$^+$07] Srinivasan Murali, Almir Mutapcic, David Atienza, Rajesh Gupta, Stephen Boyd, and Giovanni De Micheli. Temperature-aware processor frequency assignment for mpsocs using convex optimization. In *CODES/ISSS*, 2007.

[MMNBR07] Francisco Javier Mesa-Martinez, Joseph Nayfach-Battilana, and Jose Renau. Power Model Validation Through Thermal Measurements. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA, pages 302–311, 2007.

[MRL$^+$10] Timothy G. Mattson, Michael Riepen, Thomas Lehnig, Paul Brett, Werner Haas, Patrick Kennedy, Jason Howard, Sriram Vangal, Nitin Borkar, Greg Ruhl, and Saurabh Dighe. The 48-core scc processor: The programmer's view. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[MRR$^+$15] Ramya Jayaram Masti, Devendra Rai, Aanjhan Ranganathan, Christian Müller, Lothar Thiele, and Srdjan Capkun. Thermal covert channels on multi-core platforms. In *24th USENIX Security Symposium (USENIX Security 15)*, Washington, D.C., 2015. USENIX Association.

[Mü15] Christian Müller. A Framework for End-to-End Thermal Modeling and Control. Master's thesis, ETH Zurich, Zurich, Switzerland, 2015.

[N$^+$08] Hristo Nikolov et al. Systematic and Automated Multiprocessor System Design, Programming, and Implementation. *IEEE Trans. Comput. Aided Design*, 27(3):542–555, 2008.

[NMA$^+$12] M. Neukirchner, T. Michaels, P. Axer, S. Quinton, and R. Ernst. Monitoring arbitrary activation patterns in real-time systems. In *Real-Time Systems Symposium (RTSS), 2012 IEEE 33rd*, pages 293–302, Dec 2012.

[Off13] BV Offspark. Polarssl. *ht tps://polarssl. org/, last access*, 2013.

[PNP13] W. Puffitsch, E. Noulard, and C. Pagetti. Mapping a multi-rate synchronous language to a many-core processor. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 293–302, April 2013.

[PSHA03] J.J. Purcell, P. Sowerby, E.P. Hodzen, and E.B. Andrews. ECU temperature control, December 2 2003. US Patent 6,655,326.

[RSS$^+$13] Devendra Rai, Lars Schor, Nikolay Stoimenov, Iuliana Bacivarov, and Lothar Thiele. Designing Applications with Predictable Runtime Characteristics for the Baremetal Intel SCC. *Runtime and Operating Systems for the Many-core Era (ROME)*, 2013.

[RU08] R. Rao and Arizona State University. *Fast and Accurate Techniques for Early Design Space Exploration and Dynamic Thermal Management of Multi-core Processors*. Arizona State University, 2008.

[RV09] Ravishankar Rao and Sarma Vrudhula. Fast and accurate prediction of the steady-state throughput of multicore processors under thermal constraints. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28(10):1559–1572, October 2009.

[RW89] J.-C. Ryou and J.S.K. Wong. A Task Migration Algorithm for Load Balancing in a Distributed System. In *Proc. System Sciences*, pages 1041–1048, 1989.

[S$^+$05] S. Sankaran et al. The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing. *Journal of High Performance Computing Applications*, 19(4):479–493, 2005.

[SBYT12] Lars Schor, I. Bacivarov, Hoeseok Yang, and L. Thiele. Fast worst-case peak temperature evaluation for real-time applications on multi-core systems. In *Test Workshop (LATW), 2012 13th Latin American*, pages 1–6, April 2012.

[SGHM14] Muhammad Shafique, Siddharth Garg, Jörg Henkel, and Diana Marculescu. The EDA Challenges in the Dark Silicon Era: Temperature, Reliability, and Variability Perspectives. In *Proceedings of the 51st Annual Design Automation Conference*, DAC ’14, pages 185:1–185:6, New York, NY, USA, 2014. ACM.

[SGM+14] Muhammad Shafique, Siddharth Garg, Tulika Mitra, Sri Parameswaran, and Jörg Henkel. Dark Silicon As a Challenge for Hardware/Software Co-design: Invited Special Session Paper. In *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*, CODES '14, pages 13:1–13:10, New York, NY, USA, 2014. ACM.

[SMD14] Maryam Saadvandi, Karl Meerbergen, and Wim Desmet. Parametric dominant pole algorithm for parametric model order reduction. *Journal of Computational and Applied Mathematics*, 259, Part A:259 – 280, 2014.

[SS13] S. Sankaran and R. Sridhar. Energy modeling for mobile devices using performance counters. In *Circuits and Systems (MWSCAS), 2013 IEEE 56th International Midwest Symposium on*, pages 441–444, Aug 2013.

[SSS+04a] Kevin Skadron, Mircea R. Stan, Karthik Sankaranarayanan, Wei Huang, Sivakumar Velusamy, and David Tarjan. Temperature-aware microarchitecture: Modeling and implementation. *ACM Trans. Archit. Code Optim.*, 1(1):94–125, March 2004.

[SSS+04b] Kevin Skadron, Mircea R. Stan, Karthik Sankaranarayanan, Wei Huang, Sivakumar Velusamy, and David Tarjan. Temperature-aware microarchitecture: Modeling and implementation. *ACM Trans. Archit. Code Optim.*, 1(1):94–125, 2004.

[SVAB13] Arvind Sridhar, Alessandro Vincenzi, David Atienza, and Thomas Brunschwiler. 3d-ice: A compact thermal model for early-stage design of liquid-cooled ics. *IEEE Transactions on Computers*, 99(PrePrints):1, 2013.

[SW92] T.T.Y. Suen and J.S.K. Wong. Efficient Task Migration Algorithm for Distributed Systems. *IEEE Trans. on Parallel and Distributed Systems*, 3:488–499, 1992.

[TCGK02] Lothar Thiele, Samarjit Chakraborty, Matthias Gries, and Simon Künzli. Design space exploration of network processor architectures. In *First Workshop on Network Processors at the 8th International Symposium on High-Performance Computer Architecture (HPCA8)*, pages 30–41, Cambridge MA, USA, 2002.

[TCN00] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. *ISCAS*, 4:101–104, 2000.

[TGC+06] Ian A. Troxel, Eric Grobelny, Grzegorz Cieslewski, John Curreri, Mike Fischer, and Alan D. George. Reliable management services for cots-based space systems and applications. In *Proc. International Conference on Embedded Systems &Applications*, pages 169–175, 2006.

[vdWMH11] Rob F. van der Wijngaart, Timothy G. Mattson, and Werner Haas. Light-weight communications on intel's single-chip cloud computer processor. *SIGOPS Oper. Syst. Rev.*, 45(1):73–83, February 2011.

[VEMR14] Anish Varghese, Bob Edwards, Gaurav Mitra, and Alistair P. Rendell. Programming the adapteva epiphany 64-core network-on-chip coprocessor. In *Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, IPDPSW '14, pages 984–992, Washington, DC, USA, 2014. IEEE Computer Society.

[W+10] Chao Wang et al. Hybrid Checkpointing for MPI Jobs in HPC Environments. In *Proc. ICPADS*, pages 524–533, 2010.

[WB06a] S. Wang and R. Bettati. Delay analysis in temperature-constrained hard real-time systems with general task arrivals. In *RTSS*, 2006.

[WB06b] S. Wang and R. Bettati. Reactive speed control in temperature-constrained real-time systems. In *Euromicro Conference on Real-Time Systems*, 2006.

[WB08] S. Wang and R. Bettati. Reactive speed control in temperature-constrained real-time systems. *Real-Time Systems Journal*, 39(1-3):658–671, 2008.

[WH11] Jian Wang and Fu-yuan Hu. Thermal hotspots in cpu die and its future architecture. In Ran Chen, editor, *Intelligent Computing and Information Science*, volume 134 of *Communications in Computer and Information Science*, pages 180–185. Springer Berlin Heidelberg, 2011.

[WMT06] Ernesto Wandeler, Alexandre Maxiaguine, and Lothar Thiele. Performance analysis of greedy shapers in real-time systems. In *DATE*, pages 444–449, 2006.

[WMW09] Yefu Wang, Kai Ma, and Xiaorui Wang. Temperature-constrained power control for chip multiprocessors with online model estimation. In *ISCA*, pages 314–324, 2009.

[WR10] Zhe Wang and S. Ranka. A simple thermal model for multi-core processors and its application to slack allocation. In

Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on, pages 1 –11, april 2010.

[WSMM01a] D.J. Walkey, T.J. Smy, T. MacElwee, and M. Maliepaard. Linear models for temperature and power dependence of thermal resistance in si, inp and gaas substrate devices. In *Semiconductor Thermal Measurement and Management, 2001. Seventeenth Annual IEEE Symposium*, pages 228 –232, 2001.

[WSMM01b] D.J. Walkey, T.J. Smy, T. MacElwee, and M. Maliepaard. Linear models for temperature and power dependence of thermal resistance in si, inp and gaas substrate devices. In *Semiconductor Thermal Measurement and Management, 2001. Seventeenth Annual IEEE Symposium*, pages 228 –232, 2001.

[WT06a] Ernesto Wandeler and Lothar Thiele. Interface-based design of real-time systems with hierarchical scheduling. In *RTAS '06: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 243–252, Washington, DC, USA, 2006. IEEE Computer Society.

[WT06b] Ernesto Wandeler and Lothar Thiele. Real-Time Calculus (RTC) Toolbox. http://www.mpa.ethz.ch/Rtctoolbox, 2006.

[YCTK10a] Chuan-Yue Yang, Jian-Jia Chen, L. Thiele, and Tei-Wei Kuo. Energy-efficient real-time task scheduling with temperature-dependent leakage. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 9–14, March 2010.

[YCTK10b] Chuan-Yue Yang, Jian-Jia Chen, Lothar Thiele, and Tei-Wei Kuo. Energy-efficient real-time task scheduling with temperature-dependent leakage. In *ACM/IEEE Conference of Design, Automation, and Test in Europe (DATE)*, 2010.

[$Z^{+}12$] Michael Ziwisky et al. BareMichael: A Minimalistic Bare-Metal Framework for the Intel SCC. In *Proc. MARC*, pages 66–71, 2012.

[ZC07] Sushu Zhang and Karam S. Chatha. Approximation algorithm for the temperature-aware scheduling problem. In *ICCAD*, 2007.

[ZM12] C. Zimmer and F. Mueller. Low contention mapping of real-time tasks onto tilepro 64 core processors. In *Proc. Real-Time and Embedded Technology and Applications Symposium*, pages 131–140, 2012.

[Zum08] Hank Zumbahlen, editor. *Linear circuit design handbook*. Newnes, 2008.

[Åk02] Åkerholm, M. and Samuelsson, T. Design and Benchmarking of Real-Time Multiprocessor Operating System Kernels. Master’s thesis, Mälardalen University, Västerås, Sweden, 2002.

List of Publications

The following list includes publications that form the basis of this thesis. The corresponding chapters are indicated in parentheses.

D. Rai, H. Yang, I. Bacivarov, L. Thiele. **Power agnostic technique for efficient temperature estimation of multicore embedded systems.** In *Proceedings of the 2012 international conference on Compilers, Architectures and Synthesis for Embedded systems.* Tampere, Finland. (Chapter 2)

D. Rai and L. Thiele. **A Calibration Based Thermal Modeling Technique for Complex Multicore Systems.** In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015.* Grenoble, France. (Chapter 3)

D. Rai, H. Yang, I. Bacivarov, J. J. Chen, L. Thiele. **Worst-Case Temperature Analysis for Real-Time Systems.** In *Proceedings of the 2011 Design, Automation & Test in Europe Conference & Exhibition, DATE 2011.* Grenoble, France. (Chapter 5)

H. Yang, I. Bacivarov, D. Rai, J. . Chen, L. Thiele. **Real-time worst-case temperature analysis with temperature-dependent parameters.** In *Real-Time Systems.* 2013. (Chapter 5)

D. Rai, P. Huang, N. Stoimenov, L. Thiele. **An Efficient Real Time Fault Detection and Tolerance Framework Validated on the Intel SCC Processor.** In *Proceedings of the 51st Annual Design Automation Conference.* San Francisco, CA, USA. (Chapter 6)

D. Rai, P. Huang, N. Stoimenov, L. Thiele. **Distributed Stable States for Process Networks - Algorithm, Analysis, and Experiments on Intel SCC.** In *Proceedings of the 50th Annual Design Automation Conference.* San Francisco, CA, USA. (Chapter 7)

D. Rai, L. Schor, N. Stoimenov, I. Bacivarov, L. Thiele. **Distributed Stable States for Process Networks - Algorithm, Analysis, and Experiments on Intel SCC.** In *Euro-Par 2013: Parallel Processing Workshops.* Aachen, Germany. (Appendix A)

The following list includes publications that are not part of this thesis.

R. J. Masti, D. Rai, A. Ranganathan, C. Müller, L. Thiele, S. Capkun. **Thermal Covert Channels on Multi-core Platforms.** In *The 24th USENIX Security Symposium (USENIX Security '15).* Washington D.C, USA.

L. Schor, I. Bacivarov, L. G. Murillo, P. S. Paolucci, F. Rousseau, A. El Antably, R. Buecs, N. Fournel, R. Leupers, D. Rai, L. Thiele, L. Tosoratto, P. Vicini, J. Weinstock. **EURETILE Design Flow: Dynamic and Fault Tolerant Mapping of Multiple Applications Onto Many-Tile Systems.** In *Proceedings of the 2014 IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA).* Milan, Italy.

L. Schor, I. Bacivarov, D. Rai, H. Yang, S. H. Kang, L. Thiele. **Scenario-based Design Flow for Mapping Streaming Applications Onto On-chip Many-core Systems.** In *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems.* Tampere, Finland.

Curriculum Vitæ

Personal Data

Name	Devendra Rai
Date of Birth	April 6, 1981
Citizenship	Indian

Education

2010–2015	ETH Zurich Computer Engineering and Networks Laboratory Ph.D. under the supervision of Prof. Dr. Lothar Thiele
2007–2009	University of Virginia, USA Master of Science in Computer Engineering
2000–2004	National Institute of Technology, Surathkal, India. Bachelor of Technology in Electronics and Communication Engineering

Professional Experience

2009–2010	Peri Software Solutions, USA Technology Consulant
2004–2007	Delphi Automotive Systems, India/Germany Model Based Design and Engineering Software Engineer

Today, we expect that each new generation of computer processors delivers higher computational performance as compared to its predecessor. Thus, as users of recent generation of mobile phones, we can now play games with rich graphics, enjoy rich multimedia, navigate the world, create and edit movies, none of which was possible only a few years ago. Similarly, our notebook computers are now as powerful as desktops of yesterday, allowing us to work effectively on the move.

However, the impressive computational performance of state-of-the-art processors is not without its own design challenges, which must be properly understood and overcome. Specifically, such processors are increasingly vulnerable to overheating, due to non ideal technology scaling. In the long term, processors which either run too hot or experience rapid changes in temperature have reduced reliability as compared to the processors whose operating temperature has been carefully controlled. In the short term, the performance of tasks and services executing on a hot processor may deteriorate to an unacceptable level, and in extreme cases, hosted tasks and services may become unavailable if the processor shuts itself down to avoid any thermally induced damage.

This thesis presents three mutually orthogonal and complementary approaches to improving the reliability of systems and services which are based on state-of-the-art processors. Specifically, this thesis presents design tools and techniques which can be used to avoid thermally induced faults, tolerate them, and also recover from faults. Fault tolerance and recovery approaches have been specifically developed in the context of time and resource constrained systems. All approaches presented in this thesis have been tested by prototyping on state-of-the-art multi and many core processors, such as the Intel Single Chip Cloud Computer, Intel Xeon processor, and the Intel i7 mobile processor.