Minimising Access Conflicts on Shared Multi-Bank Memory

ANDREAS TRETTER, GEORGIA GIANNOPOULOU, MATTHIAS BAER,
and LOTHAR THIELE, ETH Zürich

A common multi-core pattern consists of processors communicating through shared, multi-banked on-chip memory. Two approaches exist: Interleaved address mapping, which spreads consecutive data over all banks, and contiguous address mapping, which stores consecutive data on a single bank.

In this work, we compare both approaches on the Kalray MPPA-256 platform. For contiguous mapping, we propose an algorithm, based on graph colouring techniques, to automatically perform the assignment of data blocks to memory banks with the goal of minimising access collisions and delays. Experiments with representative, parallel real-world benchmarks show that 69% of the tested configurations, when optimised for contiguous mapping by our algorithm, run up to 86% faster on average than with interleaved mapping.

CCS Concepts: • **Computing methodologies** → **Shared memory algorithms**; • **Computer systems organization** → **Embedded software**; *System on a chip*; • **Hardware** → Memory and dense storage;

Additional Key Words and Pharses: Contention, memory banks, interleaved addressing, memory allocation

ACM Reference format:
Andreas Tretter, Georgia Giannopoulou, Matthias Baer, and Lothar Thiele. 2017. Minimising Access Conflicts on Shared Multi-Bank Memory. *ACM Trans. Embed. Comput. Syst.* 16, 5s, Article 135 (September 2017), 20 pages.
https://doi.org/10.1145/3126535

1 INTRODUCTION

Communication over shared memory has become a prevalent concept in embedded multi- and many-core systems on chip. Many platforms group middle-ranged numbers of processing cores to clusters with one common memory space. To allow multiple simultaneous accesses from the different cores, these memory architectures typically consist of multiple banks with independent interfaces. Commonalities end, however, when it comes to the question of how to distribute the data between the banks, as Figure 1 illustrates: While some architectures (like the Adapteva Epiphany [23]) provide explicit access to the individual banks and leave the storage concept to the programmer (*contiguous* memory address mapping), others (like the P2012 [2] or the PULP [9]) opt for a strict word-granularity spreading of the entire data over all the banks (*interleaved* memory address mapping).

The latter option has several advantages: All the cores see a uniform memory space, in which the programmer can accommodate any code or data without having to think of its particular

This article was presented in the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES) 2017 and appears as part of the ESWEEK-TECS special issue.
Authors' addresses: A. Tretter, G. Giannopoulou, M. Baer, and L. Thiele, Computer Engineering and Networks Laboratory, ETH Zürich, 8092 Zürich, Switzerland; email: firstname.lastname@tik.ee.ethz.ch.

ACM 1539-9087/2017/09-ART135 $15.00
https://doi.org/10.1145/3126535

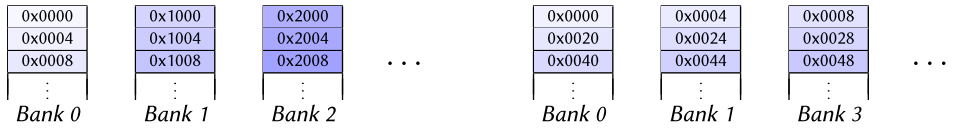


Fig. 1. Examples for different memory address mapping types. For each of the first three banks, the addresses of the first three words stored therein are shown and optically illustrated by darker colors for higher addresses.

placement. The bank access pattern of each core is essentially random and collisions approximately follow a probabilistic distribution [34]. As a result, the memory space can be treated like a single bank and the standard compilers and linkers can be used without any necessary adaptions while still achieving good performance.

At the same time, contiguous mapping has its advantages as well. Firstly, the performance of many low-latency implementations of interleaved mapping is vulnerable to memory access patterns which repeatedly access the same bank (e.g. operations on matrices having dimensions of powers of two). This can be avoided by contiguous mapping with appropriate data placement. Secondly, worst-case delays are harder to determine on interleaved memory systems, and calculated bounds are usually far from tight. This is why recent publications in that domain focus on contiguous mapping, see e.g. [1, 3, 22, 25, 36].

When average case performance is more important than worst case execution, there is no such clear preference. It would therefore be desirable to have a direct comparison between both approaches, which up to now does not exist. However, to perform such a comparison, one would also need to find a solution to the question of data placement in the case of contiguous mapping, i.e., on which bank which data should be placed. Currently, this has to be decided manually by the programmer, which constitutes a significant obstacle to the application of contiguous mapping. Finding a way of automatically partitioning data between the memory banks would thus not only allow a comparison between the different mapping approaches, but it would also demonstrate the practical feasibility of using contiguous mapping.

Several approaches and algorithms have been proposed to tackle the data placement issue for single-processor multi-bank systems, in particular VLIWs and DSPs. Unfortunately, the corresponding optimisation problem significantly differs from the multi-core problem, since compiler backends for individual cores can easily detect simultaneous memory access on instruction level. On multi-core systems, in contrast, the individual cores are independent and only loosely synchronised. Access conflicts happen non-deterministically. As a result, existing single-core solutions cannot be directly applied to multi-processor systems.

In this work, we try to close the aforementioned gaps with a two-fold contribution: Firstly, we propose a heuristic algorithm to automate data placement, such that every data block is assigned a memory bank for contiguous mapping. The proposed approach was designed to find mappings that minimise access conflicts as well as other delays, and thus the application runtime. Secondly, with the help of and as an evaluation for the newly introduced algorithm, we compare the average case performance of contiguous and interleaved memory mapping, theoretically as well as experimentally, on the Kalray MPPA platform [10]. This platform is very well-suited for this comparison, since it allows the programmer to choose between interleaved and contiguous memory mapping. To obtain meaningful results in a wider context, we have conducted dedicated, synthetic experiments on the MPPA and we have run real world benchmarks.

With a memory bank assignment optimised using the proposed algorithm, 96.7% of the tested benchmark configurations perform at least as well as with interleaved memory mapping. In 54.5% of the cases, the runtime is even significantly shorter.

In Section 3, we describe our generic models of the applications to be implemented and of the target platforms. On this basis, we formally define the bank assignment problem for the contiguous memory mapping scenario. We then introduce an algorithm for solving this generic bank assignment problem in Section 4. Section 5 gives detailed information about the MPPA and its memory architecture. Based on the results of synthetic benchmarks revealing the platform's characteristics, we show how the generic bank assignment algorithm can be adapted to this particular platform. Section 6 finally shows the results of the experimental comparison of contiguous and interleaved mapping on the Kalray MPPA, using the proposed bank assignment algorithm for the case of contiguous memory address mapping.

2 RELATED WORK

Various works address the problem of mapping data to memory banks for single-processor systems. In particular, there are many works on VLIW (in particular DSP) systems with dual-bank memories, e.g. by Saghir et al. [29], Leupers and Kotte [18], Cho et al. [6], Sipkova [31], Ko and Bhattacharyya [17] and Murray and Franke [21]. All of these works try to enable two simultaneous memory accesses by mapping the corresponding variables (or arrays) to different memory banks. While [29], [18] and [6] propose implementations as compiler backends, [31], [17] and [21] analyse the code on higher levels.

Zhang et al. [39] and Soto et al. [32] extend this optimisation to a variable number of memory banks. Conversely, Shyam and Govindarajan [30] try to minimise energy consumption in such a system by setting assigning the banks such that some of them can be brought to sleep mode as often as possible. [39] also supports this optimisation.

Most of the works discussed until now make use of so-called *conflict* or *interference graphs*, as we do in this work. The typical solution approaches are greedy algorithms, other heuristics, or integer linear programming. [32] and [21] treat the assignment task as a graph colouring problem, like it is done in this work. However, all these works consider single processor systems and rely on conflict analysis techniques that are not applicable to multi-processor systems as discussed earlier.

Kim and Kim [15] propose a method to improve performance of multiple DRAM banks connected to a single processor. Their approach is to maximise spatial access locality within each memory bank, thus avoiding costly row opening operations required on this memory architecture. In our work, we consider platforms with on-chip SRAMs, on which access locality has no influence.

In the area of multi-core systems, [16] and [20] propose heuristics for mapping data of different application threads to DRAM banks to reduce the average thread execution times. Kim et al. [16] present a compiler approach targeting coarse grain reconfigurable architectures. Our approach is more general, since it is applicable to any homogeneous shared-memory architecture. Mi et al. [20] introduce a software/hardware scheme for static DRAM bank partitioning. Purely hardware-based solutions include the works of Reineke et al. [27] and Wu et al. [37], which rely on DRAM controllers to implement bank privatisation schemes. Such approaches require special hardware, while we propose compiler techniques that are applicable to commercial off-the-shelf platforms.

Other works implement software approaches to completely eliminate bank-level conflicts. Liu et al. [19] implement a custom page-colouring algorithm inside the memory management of the Linux kernel for this purpose. Jeong et al. [14] propose a combination of bank partitioning and memory sub-ranking, implemented through an extension of the OS physical frame allocation algorithm. Yun et al. [38] implement a DRAM bank-aware memory allocator to allocate memory

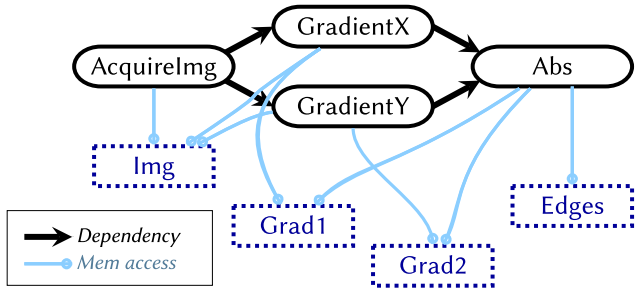


Fig. 2. Representation of a simple image edge detection algorithm in the application model from Section 3.1. It reads an image into memory, numerically computes its gradients in x and y direction and then computes the edge intensity as the euclidean norm of both. The upper part shows the tasks, the lower part the memory blocks.

pages of different applications to private banks. Chandru et al. [5] implement a user space bank-aware and controller-aware allocator, which enables binding a core to a specific bank and controller in a cluster-based architecture. Pan et al. [24] enable frame allocation on thread-specific cache, memory controller and memory bank combinations through an according modification to the Linux kernel. All these works try to cleanly (or at least largely) separate the banks accessed by the different applications or cores. In contrast, we assume that bank sharing is indispensable for communication between cores and that each core will thus need regular access to multiple banks.

Closer to our work lies the approach of Giannopoulou et al. [12], which also tries to minimise bank conflicts through optimized data-to-bank mapping. However, their method aims at minimising the worst-case execution time of real-time applications and is bound to specific scheduling policies, whereas we address the average-case execution time of a wider class of applications. Rihani et al. [28] propose a solution for single-producer single-consumer process networks by assigning each consumer a bank for all its input FIFOs. This works well for the selected application model, which, however, necessitates large amounts of data copying and is therefore inherently inefficient on shared memory platforms [35]. Finally, Goens et al. [13] employ a buffer allocation approach similar to this paper. While they use a more general platform model and a more detailed application model, this comes at the price of longer optimisation time (hours as compared to milliseconds). Also, their model does not cover the particularities of the MPPA platform (cf. Section 5.2).

3 CONSIDERED MEMORY BANK ASSIGNMENT PROBLEM

As discussed earlier, in the case of contiguous memory mapping, each buffer or piece of data accessed by the application must be assigned to a memory bank on which it will reside. This assignment should be done in such a way that interferences between different threads are minimised. This section gives a detailed definition of the problem, showing how we model the application and the target platform. All models are kept as generic as possible, and an algorithm to solve the assignment problem for the generic case will be given in the next section. Later sections will then discuss how the generic models may or may need to be adapted to concrete target platforms such as the Kalray MPPA.

3.1 Application Model

Figure 2 illustrates how we model the applications that are executed on a cluster.

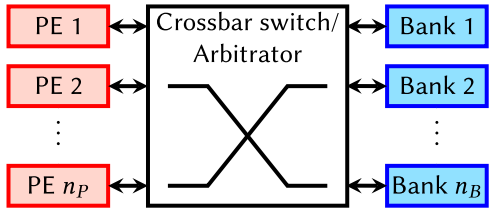


Fig. 3. Exemplified illustration of the generic type of platform considered in this work.

Definition 3.1. An **application** is a tuple (T, D, M, a) of a set $T = \{t_1, \ldots, t_{n_T}\}$ of **tasks**, a set $D \subset T \times T$ of **dependencies**, a set $M = \{m_1, \ldots, m_{n_M}\}$ of **memory blocks** and an **access function** $a : T \to \mathcal{P}(M)$.

Each task t executes exactly once and accesses only the memory blocks contained in $a(t) \subseteq M$. Each dependency $(t_i, t_j) \in D$ enforces that t_j can only start once t_i has finished. Each memory block $m \in M$ has a distinct **size** $s(m)$, with $s : M \to \mathbb{N}$.

This model resembles process networks or dataflow models, where dependencies determine the (partial) execution order of the tasks. Dependencies do, however, not signify data exchange. Instead, all data operations are modelled as accesses to explicitly specified memory regions – the blocks. Copying of data does not take place unless it is modelled as a task in the application. The purpose of modelling dependencies is only to ensure correct execution order, to prevent race conditions and to guarantee the correctness of the results. Note that no assumptions are made as to how or in what pattern a task t accesses the memory blocks given by $a(t)$.

The model is a simplified representation of the OpenVX execution model [11] and the Memory Interference Graph [12], but it is intentionally generic to be applicable to a broad class of different execution models, for instance mechanisms like Deterministic Memory Sharing in Kahn Process Networks [35] or traditional custom multi-threading with manual thread synchronisation.

In this work, we use the concept of memory blocks to model data inputs and outputs of the tasks as well as local data/intermediate storage. We do not model instruction fetches with it. While the latter could be done as well without changing the methods we propose, we consider these effects to be negligible. For instance, MPPA cores can draw on instruction caches that are large enough to hold the complete code required for a typical task. On multi-core platforms, these tasks typically consist of loops, which are once loaded into the cache and then executed a large number of times without any further instruction fetch from memory.

3.2 Platform Model

Figure 3 gives an (abstract) example of the kind of target platform considered in this work. It can be formalised in the following, even more generic model.

Definition 3.2. A **platform** is a tuple (P, B, c), with $P = \{p_1, \ldots, p_{n_P}\}$ a set of processing elements (PEs) that share access to a set of memory banks $B = \{b_1, \ldots, b_{n_B}\}$. $c : B \to \mathbb{N}$ gives the capacity of each memory bank.

We make the following assumptions about a platform:

- The banks are configured for contiguous address mapping.
- Each PE takes the same time to access each bank.
- All banks can be accessed in parallel, but each bank can only be accessed by one PE at a time, e.g. by a crossbar communication structure between PEs and banks.

- The arbitration between bank accesses is fair, e.g. Round Robin.
- The PEs do not perform task preemption.

This model fits to many existing multi-core multi-bank platforms, for instance the NXP LPC family or the Kalray MPPA (details will be given later). It would fit to the P2012 and the PULP architectures as well if they supported a contiguous memory address layout. Also, it comes very near to platforms like the Adapteva Epiphany chips. Since the latter have a non-uniform memory access architecture (different access times to different banks), certain adaptions would be necessary, but entirely feasible.

3.3 Problem Description

The problem to be solved now is assigning memory blocks to banks. With the previous definitions, it can be described as follows.

Let (T, D, M, a) be an application to run on a platform (P, B, c). Let there further be a given mapping $T \to P$ and a schedule for execution of the tasks.

Find a mapping $f : M \to B$ that assigns each memory block to a bank such that:

- All memory blocks fit into the banks they are assigned to, i.e. for each $b \in B$

$$\sum_{m \in M_b} s(m) \leq c(b) \quad \text{with} \quad M_b = \{m \in M \mid f(m) = b\}.$$

- The time between the execution start of the first and the execution end of the last task in the schedule is minimised. In this context, "time" denotes average values, as run-times can vary due to synchronisation and data dependencies, for example.

4 GENERIC MEMORY BANK ASSIGNMENT ALGORITHM

This section describes the approach proposed for solving the problem defined in the previous section, i.e., for assigning memory blocks to memory banks in the generic case with the generic platform model. First, the challenges of conceiving such an algorithm are discussed, and a general overview of the proposed method is given. Then the algorithm itself is presented and simple optimisations are discussed.

4.1 Overall Approach

A simple idea for solving the problem might be a heuristic that follows the idea of static *load balancing*, i.e., that distributes the blocks evenly among the banks. This heuristic would assign banks to all blocks in the order of their size, starting from the largest block. For each block, it would select the bank which, at the current state of assignment, has the most free space. Such an algorithm attempts to find a valid bank assignment if one exists and to distribute the access bandwidth over the banks. However, it is fully agnostic to program semantics and cannot detect possible access collision hotspots. For instance, in the application example from Figure 2, the most promising optimisation would be to assign `Grad1` and `Grad2` to different banks. With a load balancing algorithm, however, in case of resource scarcity, whether this happens or not is coincidence.

A good bank assignment algorithm therefore needs to model possible access collisions between two tasks. These depend on multiple factors, such as whether the execution times of the tasks overlap, how often they access the memory blocks, in what pattern they do so, etc. Note that there exist cyclic dependencies between the timing of the tasks (overlaps) and the bank assignments. Also, there may be conflicting optimisation criteria on particular platforms. For instance, in order to avoid access collisions between different cores on the MPPA platform, one would often like to place memory blocks on different banks. On the other hand, if these blocks are later accessed

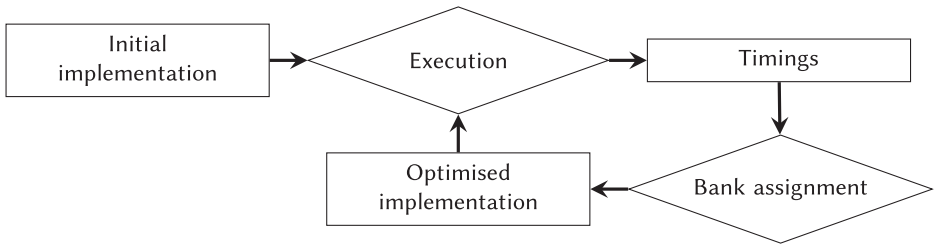


Fig. 4. Flow chart diagram of the iterative approach proposed for assigning memory blocks to banks.

simultaneously by one core, it may be advantageous to place them on the same bank, as will be shown later.

For modelling access collisions and the aforementioned related problems, many different analysis methods have been proposed and could be successfully applied here. For reasons of simplicity, however, we choose a heuristic approach: We take the time that two tasks execute in parallel as an indication for the occurrence of access collisions on their associated memory blocks. This time is determined by measurements. By cumulating the execution time overlaps from all pairs of tasks, we obtain pairwise access conflict potentials between all the memory blocks. Whether these conflicts materialise depends on the bank assignment of the blocks, which we try to optimise.

ALGORITHM 1: Graph Colouring Based Bank Assignment.

input : Application (T, D, M, a), platform (P, B, c),
task execution times $start : T \to \mathbb{N}, end : T \to \mathbb{N}$
output : Bank assignment $colours : M \to B$
$(V, E) \leftarrow$ CreateCollisionGraph$((T, D, M, a), start, end)$
for $i \leftarrow 1 \ldots |M|$ **do**
  $R_i \leftarrow$ SelectRemoveCandidate$((V, E), (T, D, M, a))$
  $(V, E) \leftarrow$ RemoveNode(V, E, R_i)
for $i \leftarrow |M| \ldots 1$ **do**
  $(V, E) \leftarrow$ ReinsertNode(V, E, R_i)
  $colours[R_i] \leftarrow$ ChooseColour$((V, E), (P, B, c), colours, R_i)$

This leads to the iterative approach shown in Figure 4. First, the application is executed in an initial implementation, measuring the start and finish times of all tasks. This initial implementation could either be with interleaved memory address mapping if the platform allows it, or contiguous memory mapping with a bank assignment obtained by a simple heuristic like the load-balancing based method mentioned previously. Based on these times, a bank assignment is obtained by our algorithm. Application execution with this new assignment yields new timings, which are then used for a refined bank assignment as the execution leads to different assumptions on overlapping accesses to shared memory banks. This is continued until the assignment converges or for a fixed number of iterations. In our experiments, ten iterations proved to be sufficient to get a steady-state behaviour or a limit cycle.

The following section will present the above described algorithm that assigns banks to memory blocks.

4.2 Basic Bank Assignment Algorithm

As already mentioned, we need an algorithm that, given the execution times for all tasks from an application, finds a bank assignment that minimises access conflicts between different tasks.

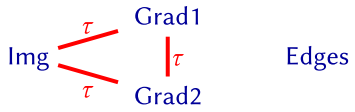


Fig. 5. Conflict graph for the sample application from Figure 2.

To attain this goal, we express the problem as a graph colouring problem. In the graph to be coloured, each memory block is represented by a node and an edge is inserted between two nodes when the corresponding memory blocks are accessed in parallel by different tasks. The banks are represented by colours, so the number of colours is fixed. A fundamental difference to classic graph colouring is that the latter does not allow two neighbouring nodes to have the same colour. For bank assignment, this case would mean possible access conflicts on the concerned bank, but the program would still execute correctly. It is therefore allowed but will yield a performance penalty.

The algorithm has been inspired by a well known graph colouring based solution to the register allocation problem [4] and is shown in Algorithm 1. It consists of constructing a *collision graph*, gradually removing all nodes from it and re-inserting and colouring them in reverse order. The idea of the node removal phase is to fix an order in which the nodes are coloured: Since each colouring decision reduces the degrees of freedom for the remaining nodes, it is important on the one hand to colour those nodes first for which conflicts would have the greatest impact and on the other hand those that have a low degree of freedom already from the start. The details of the different phases shall be layed out in the following.

ALGORITHM 2: Function CreateCollisionGraph

input : (T, D, M, a), $start$, end
output : Collision graph (V, E)
$(V, E) \leftarrow (M, \{\text{pairs}(M) \rightarrow 0\})$
foreach pair $t_1 \neq t_2 \in T$ **do**
  $o \leftarrow \min(end(t_1), end(t_2)) - \max(start(t_1), start(t_2))$ //*calculate execution time overlap o*
  if $o > 0$ **then**
    foreach $m_1 \in a(t_1), m_2 \in a(t_2), m_1 \neq m_2$ **do**
      $E[\{m_1, m_2\}] \leftarrow E[\{m_1, m_2\}] + o$

(a) Construction of the collision graph: Given an application (T, D, M, a), the collision graph is an undirected, weighted graph with the memory blocks in M as the nodes. Its construction (function CreateCollisionGraph) is given in Algorithm 2. For each pair of tasks $t_1 \neq t_2 \in T$ that execute in parallel, an edge is inserted between each pair of blocks $m_1 \in a(t_1), m_2 \in a(t_2), m_1 \neq m_2$. These are the blocks that are accessed in parallel by both tasks. The weight of the edge is equal to the time that t_1 and t_2 execute in parallel and multiple edges between the same nodes are combined to one edge with the sum of the weights.

Figure 5 shows the conflict graph for the case of the example application from Figure 2. It is assumed that `GradientX` and `GradientY` execute in parallel for τ cycles. As a result, the graph contains three edges between the blocks `Img`, `Grad1` and `Grad2`, each with the weight τ.

(b) Node removal: In this step, repeatedly a node will be chosen and removed from the graph, until the graph is empty.

The node to be removed from is determined by SelectRemoveCandidate, which can be described by the function

$$r((V,E),(T,D,M,a)) =$$
$$\arg\max_{v\in V}\left[\lambda^3\cdot|\{t\in T|v\in a(t)\}| - \lambda^2\cdot|\{v'\in V|E[\{v,v'\}]>0\}| - \lambda\cdot\sum_{v'\in V}E[\{v,v'\}] - s(v)\right]. \quad (1)$$

In this notation, λ is considered to be a symbolic constant that is very large compared to all other numbers in the formula. The function will thus return that node $v \in V$ for which the λ^3 term is the largest. Only if there are multiple nodes with the same λ^3 term, the node with the smallest λ^2 term will be returned (smallest because of the negative sign). Only in case of another draw will lower power terms of λ be taken into consideration.

The function can be described as follows. It selects the nodes to be removed *first* and later coloured *last*, i.e., the nodes with *lower priority*. For this purpose, it performs a multi-criteria comparison, where the first criterion is the most important and later ones are only considered in case of a tie. The criteria for this comparison of the nodes are explained below, ordered from high to low priority. The node to be removed is that node with

The highest number of tasks accessing the block. The reasoning behind this somewhat counter-intuitive criterion is that blocks accessed by only few tasks yield a high optimisation potential as opposed to blocks accessed by many tasks, which are anyway susceptible to access collisions.

The lowest number of neighbours. This is known as an important criterion also in register allocation, essentially because a high number of neighbours means a lower degree of freedom when trying to avoid conflicts. Therefore, we try to colour nodes with many neighbours first to still have a higher number of colours left for them. Note that the removal of nodes influences the number of neighbours left for the other nodes, often uncovering further nodes with high degrees of freedom. This is the essential idea behind the removal procedure.

The lowest cumulative weight of all adjacent edges. This is again an indicator for the optimisation potential of a block.

The lowest size of the memory block. This is only a minor criterion meant to improve the algorithm reliability (cf. Section 4.3).

For the example graph from Figure 5, this would result in `Edges` being coloured first, then `Grad1` and `Grad2`, then `Img`. While this seems unintuitive for this simple example application, it does make sense in more complex scenarios with more simultaneous tasks and more memory blocks involved. If a node (like `Edges` in this example) has no neighbours in the conflict graph, there is a chance that it can be accessed by the corresponding tasks without any conflicts. Colouring such a node first increases the probability that this chance is exploited.

(c) Node re-insertion and colouring: The graph is reconstructed by re-inserting and colouring the nodes in the reverse order of their removal before. The colour for a node and thus the bank assignment of a memory block is given by ChooseColour, which can be described by the function

$$x((V,E),(P,B,c),colours,v) = \arg\max_{b\in B^*(v)}\left[-\lambda\cdot\sum_{v'\in V|colours[v']=b}E[\{v,v'\}]+\text{free}(b)\right], \quad (2)$$

where

$$\text{free}(b) = c(b) - \sum_{v\in V|colours[v]=b}s(v)$$

gives the free space on b and $B^*(v) = \{b \in B \mid \text{free}(b) \geq s(v)\}$ is the set of banks with enough free space to accommodate memory block v. λ is again used like in (1).

This is again a multi-criteria comparison, in which those banks are chosen that have (in that order):

The least sum of weights on adjacent edges leading to nodes of the same colour. As discussed previously, we regard the weights of the edges as an indicator for the occurrence of simultaneous accesses and thus for the conflict potential between two memory blocks. If we decide to assign two adjacent nodes to the same bank, the conflict potential of the connecting edge will materialise.

The most space left on the bank. The idea behind this criterion is load balancing.

If there is no bank with enough space left to accommodate a block, the algorithm fails.

For the conflict graph from Figure 5, if we assume that only two memory banks are available, the colouring would take place as follows.

(1) `Edges` would be assigned to any of the two banks; in the following, we assume it is to the first one.
(2) `Grad1` would be assigned to the second bank, since the latter has more free space left.
(3) `Grad2` would be assigned to the first bank, since `Grad1` is assigned to the other one and there is an edge between the two nodes in the conflict graph.
(4) `Img` has edges to `Grad1` and `Grad2`, each of which is mapped to one of the two banks. Therefore, the penalty is equally high for both banks, and `Img` is assigned to the second bank, again because of free space.

If there were three banks, `Edges` and `Img` would be mapped to one bank, `Grad1` and `Grad2` to the other two. This would completely avoid access conflicts.

4.3 Improving Algorithm Reliability

The algorithm as described above fails if at one point no bank has enough space left to accommodate the block to be assigned. This can happen if smaller blocks are distributed first, not leaving sufficient space for the larger blocks. In this case, the algorithm is re-run, with a special *correction factor* γ. For this purpose, an additional term $\lambda^3 \cdot \gamma \cdot s(v)$ is added in (1). It enforces a higher priority for the block size during the node removal phase. γ is small in the beginning, but if the algorithm fails again, it is increased exponentially until bank assignment succeeds. This drastically reduces the rate of algorithm-induced failures. In our experiments, they were no longer an issue.

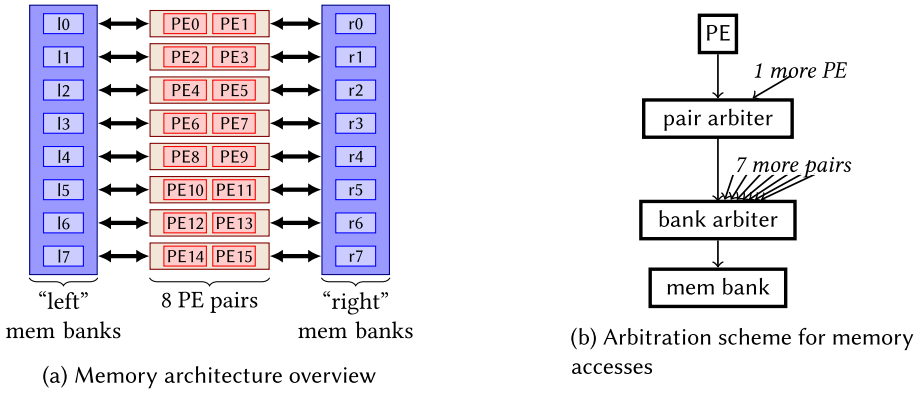
5 PLATFORM SPECIFIC ADAPTIONS

The previous sections presented a memory bank assignment approach for a generic platform. This section now shows how the method proposed earlier can be adapted to concrete platforms, which may slightly deviate from the assumptions made before.

The Kalray MPPA is going to be used as an example for such a platform. First, its architecture is described and some of its characteristics are derived from experiments. In particular, the effects of interleaved vs. contiguous memory address mapping and those of the cache will be examined. Then, the generic bank assignment algorithm is adapted to deal with the platform's special properties.

5.1 MPPA-256 Memory Architecture

The Kalray MPPA-256 Andey processor [10] integrates 256 PEs, which are grouped into 16 compute clusters. All PEs implement the same VLIW architecture.



(a) Memory architecture overview

(b) Arbitration scheme for memory accesses

Fig. 6. Illustrations of an MPPA cluster.

Figure 6(a) illustrates the architecture of a cluster. It has a local on-chip memory with 2 MB capacity, which is organized in 16 independent SRAM banks. These are arranged in two sides (*left* and *right*). The PEs are organised in 8 *pairs*. Each pair has two memory buses (one for each side), which can be utilised in parallel by the two cores. Figure 6(b) shows the arbitration hierarchy for a PE that wants to access a certain memory bank. A first conflict arises when the other core of the pair wants to access a memory bank from the same side. A second possibility for a conflict is that another pair tries to access the same bank. All these conflicts are resolved by round robin arbitration.

The memory model is von Neumann; however, each core has two private, two-way set associative caches, one for instructions and one for data. While the instruction cache is always enabled, the data cache can be enabled or disabled as needed by the programmer. Cache coherency is not supported and is a responsibility of the programmer.

Within a compute cluster, the memory address mapping can be configured either as interleaved or as contiguous. In the contiguous mapping, each bank spans 128 kB consecutive addresses. In interleaved mode, the data is distributed over all banks with a granularity of 64 byte blocks. The blocks are placed on left and on right banks in alternation.

5.2 Synthetic Memory Benchmarks

The MPPA platform has a number of properties that are important to know for achieving optimal performance in a cluster. The following experiments will demonstrate these properties.

In a first experiment, a single core reads 128 bytes individually from memory. The addresses of the bytes increase linearly with a constant stride. Figure 7 shows the time needed for reading all the bytes, as a function of the stride. The experiment was conducted with caches enabled and disabled, and with interleaved and contiguous memory configuration. The contiguous configuration shows the expected behaviour, i.e., the access time is constant and independent of the target bank. In particular, since SRAM is used, the access time is constantly low also for bigger strides. The interleaved configuration, in contrast, needs some explanation. With a stride of one byte, 64 consecutive accesses are made to the same bank before the core needs to read from a different bank. With a stride of 64 bytes, each access is to a different bank. A delay for switching banks thus explains the different timings. Assuming an additional delay caused by switches between bank sides (left to right or vice versa), one can explain the lower runtime for strides of 128 bytes, which always access banks on the same side. Overall, the measurements from this experiment perfectly

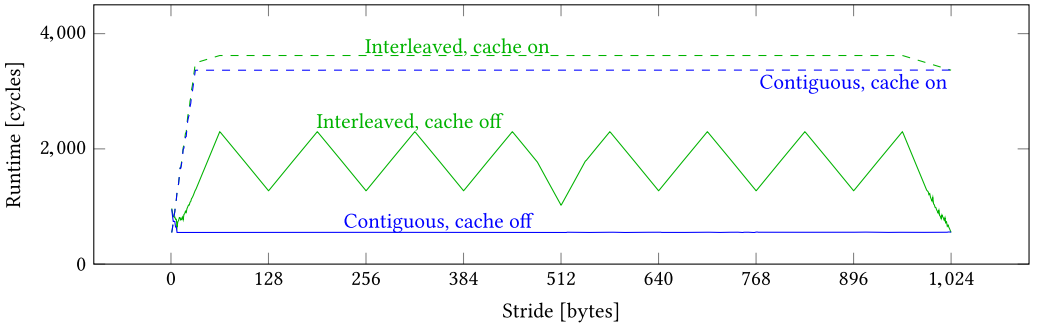


Fig. 7. Runtime for reading 128 bytes with different stride.

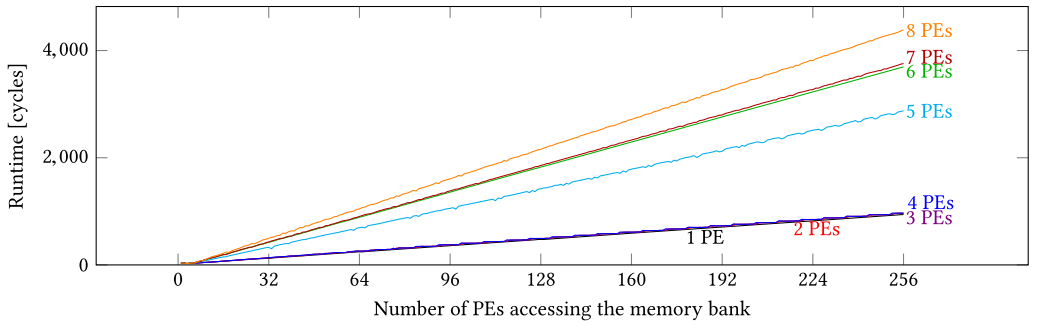


Fig. 8. Runtime for consecutive memory accesses with contention.

match a model that assumes 4 cycles for one access, 3 cycles for switching banks and 4 additional cycles for switching the side.

Another finding is that in this experiment, enabling the cache clearly worsens performance. The reason for this is that since there is no memory access locality in the code, the caches do not bring a benefit, but on the contrary, with increasing strides, lead to continuously loading entire 32 byte cache lines for only one byte that is accessed. This effect is not limited to this particular experiment but could occur for any code with little locality.

Contention effects between multiple cores have been evaluated in a second experiment. As before, one core reads data from a buffer and measures the time needed for it in contiguous mode. However, between 0 and 7 other cores access the same memory block simultaneously such as to create contention. Note that simultaneous accesses from more than 8 cores cannot occur in an MPPA cluster due to the PE pair architecture. Figure 8 shows the run-times depending on the amount of data read for different numbers of contending cores. While for less than five cores in total, the effects of contention are not significant, a large impact can be observed for more cores.

These results lead to the following general conclusions:

- For code with little memory access locality, enabling data caches can substantially decrease performance.
- With ideally partitioned data, the performance potential is higher for contiguous than for interleaved mode.
- In interleaved mode, due to the bank switching delays, even programs with data-independent control flow can have data-dependent execution times when they have memory accesses at data-dependent addresses.

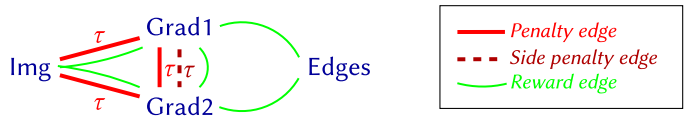


Fig. 9. Conflict graph for the sample application from Figure 2, with the MPPA-specific algorithm extensions.

With regards to contiguous memory organisation, these rules of thumb can be derived: (i) Simultaneous accesses from large numbers of cores to the same bank should be avoided. (ii) All memory accesses of one core should, if possible, be on a single memory bank to avoid bank switching delays. If multiple banks need to be used, all of them should be on the same side (left/right) to at least avoid side switching delays. (iii) The two PEs in a pair should access banks from opposing sides (this follows directly from the architecture).

5.3 Adaption of the Bank Assignment Algorithm

From the point of view of the high-level architecture, an MPPA cluster, when configured for contiguous memory layout, clearly fits the generic platform model described earlier in Section 3.2. Looking at the results of the previous experiments on the platform, however, some deviations from the assumptions made there can be spotted. The main reasons for these deviations are the organisation in PE pairs and the delays for switching banks or sides in consecutive memory accesses. These architectural specifics lead to the fact that when the system is in a particular state, accesses to some banks will take longer than to others. This, however, contradicts the assumption made before that each PE takes the same time to access each bank.

While the generic bank assignment algorithm can still be used for the MPPA platform, its results will not be optimal, since it does not take account of the platform's particularities. To obtain better assignments, the algorithm therefore needs to be adapted. The following paragraphs will show how this can be achieved.

The adapted conflict graph, again for the sample application from Figure 2, is given in Figure 9. It is assumed that the task `GradientY` is mapped to the second PE of a PE pair, all other tasks to the first PE of the same pair.

(a) Collisions within PE pairs: The two PEs in a pair should not access two banks from the same side simultaneously. To achieve this, we add a second type of weighted edges, the *side penalty edges*. Their weight is calculated like the weights of the other edges, except that only those task pairs mapped to the two PEs of a PE pair are taken into consideration. In Figure 9, there is only one such edge between `Grad1` and `Grad2`.

During the node re-insertion step, these weights are summed up for both bank sides, resulting in two side penalties w_{right} and w_{left}. The edge weight sums for each bank are then complemented with the corresponding side penalty, which corresponds to adding an additional term $-\lambda \cdot w_{\text{side}(b)}$ in (2).

(b) Banks accessed by one task: Switching between different banks takes time, so the memory blocks accessed by a task should be distributed over as few banks as possible. We approach this demand by inserting another type of (unweighted) edges in the collision graph, the *reward edges*. Such edges are inserted for each task; they are inserted between all the memory blocks it accesses. In Figure 9, the reward edges connected to `Img` come from the tasks `GradientX` and `GradientY`. All other edges come from the task `Abs`.

The reward edges are only considered in the node re-insertion step as the second criterion after the other edges: If two banks have the same sum of weights, the bank with the higher number of reward edges to adjacent blocks of the same colour is chosen. This corresponds to extending (2)

with an additional term $+\lambda^{0.5} \cdot n_{\text{reward}}(v, b)$, where $n_{\text{reward}}(v, b)$ is the number of reward edges between node v and other nodes with colour b.

(c) Bank sides accessed by one task: Switching between banks takes even more time if they belong to different sides. Therefore, if a task needs to access multiple banks, these should be on the same side (left or right). We account for this in the node re-insertion step when a node is to be coloured: We count the number of adjacent reward edges going to nodes assigned to left banks ($n_{\text{reward, left}}$) and those going to nodes assigned to right banks ($n_{\text{reward, right}}$). The results are added (with a weighting factor ω) to the reward edge count for each bank of the corresponding side. This corresponds to adding an additional term $+\lambda^{0.5} \cdot \omega \cdot n_{\text{reward, side}(b)}$ in (2). Empirically, $\omega = 0.375$ has turned out to be a good choice.

(d) Cache indices: Since the data cache on the MPPA is two-way set associative, no more than two memory blocks accessed by the same task should have similar cache indices. Otherwise, if the blocks are accessed in alternation, frequent cache misses would occur. For this reason, a mechanism that aligns the memory blocks within each bank was added as a second step after the bank assignment. It adds free spaces between the memory blocks such that the base addresses of all memory blocks accessed by a task have different cache indices. Again, an algorithm based on graph colouring is used for this purpose. The nodes are the memory blocks as before, edges are inserted between two blocks if they are used by the same task, and the colours are given by all possible cache indices. In the node removal phase, those nodes are removed first (and later coloured last) that have the most free space on the banks they have been assigned to. The nodes are reinserted filling banks from their base address upwards with memory blocks. Cache indices of the blocks are adjusted by placing "gaps" between the blocks. These gaps must be small enough to still fit into the bank.

6 PERFORMANCE COMPARISON OF INTERLEAVED VS. CONTIGUOUS MAPPING

To compare application performance of interleaved and contiguous memory configurations, we executed different applications on one cluster of a Kalray MPPA developer board, model Andey, using toolchain 1.4.2, with a bare-metal configuration. The applications comprise six different, parametrised benchmarks:

- Matrices of different sizes were multiplied, one matrix per core. Powers of two were chosen as the matrix dimensions in some cases (denoted as "matrix2") and other, random numbers in other cases (denoted as "matrix").
- The Fast Fourier Transform (FFT) of different signals in different lengths was calculated using a benchmark from [33], one FFT per core.
- A Canny edge detection filter was applied to images of different sizes. The images were split into different numbers of blocks, with one PE working on one block.
- Using convolutional neural networks taken from the CConvNet library [7, 8], up to four hand-written digits were simultaneously recognised out of images, with four PEs working together for one recognition.
- Floating point number arrays of different sizes were sorted using a merge sort algorithm. All involved cores worked in parallel for one array.
- Sparse matrices of different shapes and sizes were multiplied with dense vectors (experiment denoted as SpMV). The matrices were split into different blocks with the same size of non-zero entries, with one PE per block.

Variation of the mentioned parameters as well as of the number of active PEs (between 2 and 16) yielded a total of 345 different configurations. Note that some of the benchmarks have regular

memory access patterns (e.g. Canny), while others show data-dependent control flow (e.g. merge sort or SpMV). Usually, the former is advantageous for interleaved address mapping.

For all the benchmarks, the tasks were assigned to the different PEs by hand following regular patterns. Dynamic scheduling was used. To exclude performance influences of the binding of the tasks to the PEs, the latter was kept constant in that the same task was always assigned to the same PE in all configurations of each benchmark. Bank assignment optimisation, code generation and benchmarking were conducted in an automated process. Each configuration was implemented with the local data-caches enabled and disabled. All benchmarks were executed in three ways:

- With interleaved mode,
- With contiguous mode, optimised using the algorithm proposed in this paper,
- With contiguous mode and a bank assignment obtained using the load balancing based approach discussed in Section 4.1.

The results shall be presented in the following.

Figure 10(a) compares the run-times of the benchmarks in contiguous mode to those in interleaved mode, for the optimised bank assignment algorithm proposed in this work as well as for the load-balancing based solution. Data caches are disabled. Using the assignments determined by the proposed algorithm, the applications run equally fast or faster with a contiguous memory configuration for 95.1% of all configurations. The speed-ups are significant (more than 5%) for 75.1% of all configurations. Only in the Canny experiment, 14% of configurations performed worse with contiguous mapping; in the worst case, the runtime was 9.0 % longer than with interleaved mapping.

Figure 10(b) shows the results of the same experiments with the data caches enabled. In this case, 87.8% of all configurations run at least equally fast with contiguous memory, while speed-ups are significant in still 20.3% of the configurations. A moderate performance degradation can be seen for the SpMV benchmark. It is due to the fact that all the PEs access with a highly irregular pattern the vector to be multiplied with the sparse matrix. This results in a 14.4% longer runtime in the worst case.

Deeper insight into one of the different configurations is provided in Figure 11 in an exemplary way for the Canny experiment. The figure shows details about the absolute runtimes for one particular configuration, but with a varying number of active PEs. With enabled data cache, in-built performance monitoring counters of the MPPA platform were used to measure the number of stall cycles due to cache misses as a reference of the memory access overhead. Since the work is split between the active PEs, an increasing number of the latter yields a smaller part of the memory to be accessed by each PE; as a result, the caches get more efficient.

Both with caches enabled and disabled, the matrix2 and fft benchmarks perform significantly worse in interleaved mode. This is because the Kalray MPPA uses **sequential interleaving**, i.e., the bank a memory address is assigned to is given by a number of lower-order bits of the address. Consecutive accesses with address offsets of powers of 2 (as they occur frequently in the applications mentioned before) therefore all go to the same bank, in the case of a collision causing multiple subsequent collisions as well. This problem is well known and can be solved by **pseudo-random interleaving** [26]. While the latter technique is frequent with off-chip memory systems, to our best knowledge no on-chip memory architecture is organised in this fashion. One can conclude from this that sequential memory interleaving is also not the ideal solution for memory-agnostic programming.

Figure 12 compares the run-times with enabled caches to those with disabled caches. In 86.9% of the configurations, enabling the cache yields a speed-up. Exceptions are the "matrix2" benchmarks and few configurations of the FFT benchmark; these applications only show little locality. For all

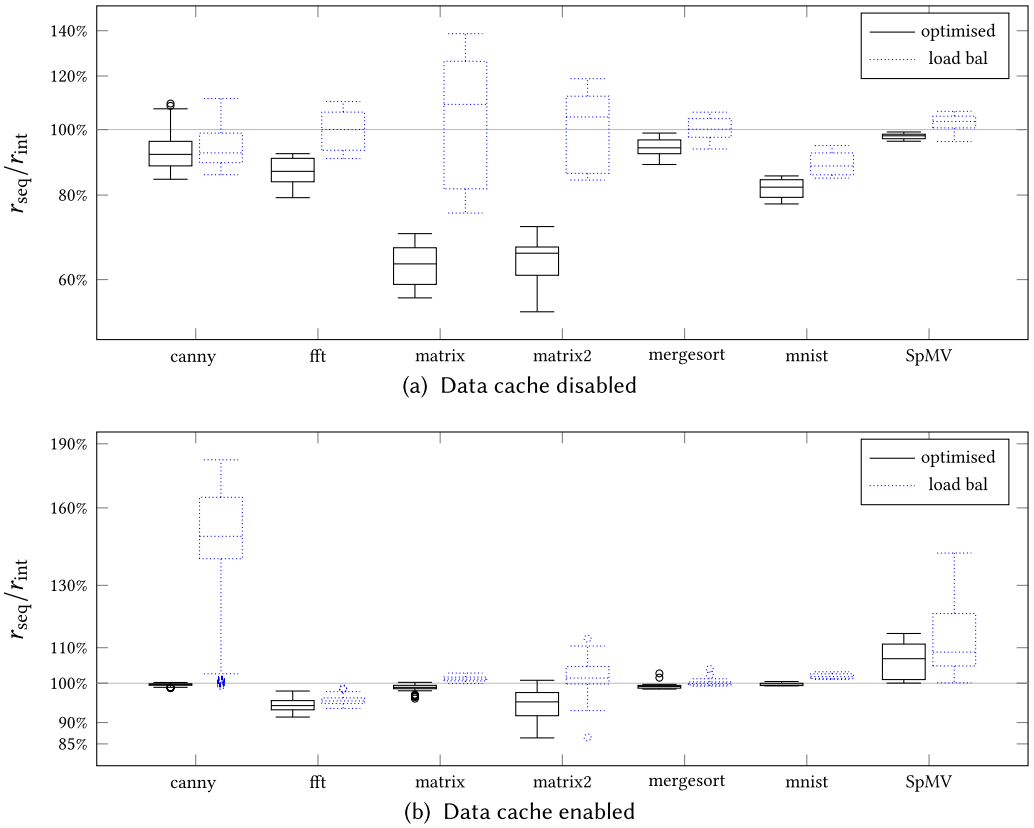


Fig. 10. Ratio of runtime in contiguous mode (r_{seq}) and runtime in interleaved mode (r_{int}), with data cache enabled or disabled. The results are shown as Tukey box plots: Each box describes their distribution for all different configurations of the corresponding benchmark. The bounds of the box mark the first and third quartiles, the band inside marks the median value. The whiskers extend to the most extreme results still within 1.5 times the inter-quartile range from the box. Results outside that range are marked separately as outliers.

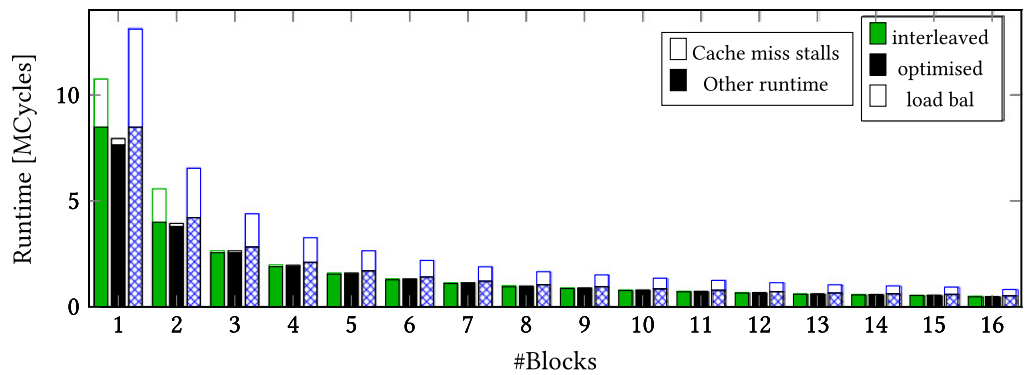


Fig. 11. Measured runtime and cache miss stalls of the Canny benchmark for a 338×258 pixel image, depending on the number of blocks it was split into (i.e. the number of PEs involved). Cache was enabled.

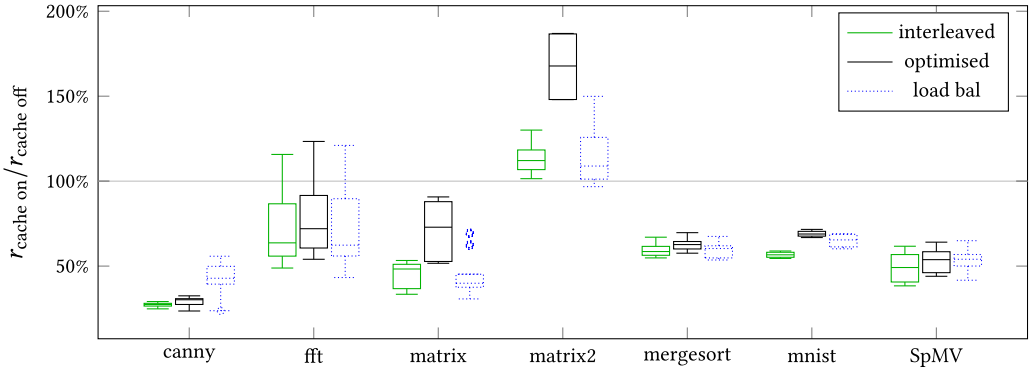


Fig. 12. Ratio of runtimes with cache enabled and disabled. The ratio is shown for interleaved mapping, for contiguous mapping optimised with the proposed algorithm and for contiguous mapping with the simple, load-balancing based method. The results are again shown as Tukey box plots like in Figure 10. For most configurations, this ratio is below 100 %, i.e., the runtime is shorter with cache enabled. One can also see that the speed-up potential of the cache is lower for configurations that already profit from an optimised bank assignment when the cache is disabled (cf. Figure 10(a)).

configurations, it holds that enabling the cache yields a speed-up either in both contiguous and interleaved mode or in none of them. The figure also shows that the optimisation potential of enabling the cache in general is lower for contiguous memory with optimised bank assignment. This is because in interleaved mode, memory accesses have a higher cost due to performance-degrading patterns like switching between different banks. Since the cache reduces the number of memory accesses, its impact is determined by that cost.

In summary, it can be stated for many applications that while the cache is able to hide certain issues like access conflicts or bank switch latencies, still better performance can be attained by avoiding these problems altogether through the choice of an appropriate, contiguous memory mapping. Particularly, in cases of low locality, only contiguous mapping boosts performance.

Independently of the aspect of performance improvement, contiguous mapping allows for better static analysability [1, 3, 22, 25, 36]. As the results show, this analysability comes at a very moderate price, the possible average performance degradation being low in most of the cases.

The runtime of the mapping algorithm itself was in the order of tens of milliseconds on an Intel Core i7-3820 CPU based machined clocked at 3.60 GHz. This makes it very short as compared to the overhead for code generation, compiling the code for the target platform etc.

7 CONCLUDING REMARKS

In this work, we compared contiguous and interleaved memory configurations on the Kalray MPPA. Synthetic benchmarks showed important characteristics for the memory access delays on this platform.

For the contiguous memory configuration, we presented a relatively simple optimisation algorithm for assigning memory banks to memory blocks. This algorithm only needs little information about the application; in particular, fine-grained profiling or in-depth static analysis are not required. Still, in real-world benchmarks we were able to attain speed-ups of up to 86 % as compared to the interleaved configuration, while significant degradation in speed only was only observed in a minority of the cases.

This shows that using contiguous memory mapping is a worthwhile alternative to interleaved mapping. With the presented algorithm, it is feasible in terms of programming effort and on the MPPA it is at least comparable in terms of average performance. Being clearly better suited for worst-case timing analysis is what makes it particularly attractive.

At the same time, the potential of this optimisation method has not been fully exploited yet. In particular, the algorithms could be clearly enhanced by adding static analysis, for instance for determining the actual accesses the tasks perform to memory blocks.

ACKNOWLEDGMENTS

This work was supported in part by the UltrasoundToGo RTD project (no. 20NA21 145911), evaluated by the Swiss NSF and funded by Nano-Tera.ch with Swiss Confederation financing.

REFERENCES

[1] Matthias Becker, Dakshina Dasari, Borislav Nicolic, Benny Akesson, Vincent Nelis, and Thomas Nolte. 2016. Contention-Free Execution of Automotive Applications on a Clustered Many-Core Platform. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE. https://doi.org/10.1109/ecrts.2016.14.

[2] L. Benini, E. Flamand, D. Fuin, and D. Melpignano. 2012. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *Design, Automation & Test in Europe Conference & Exhibition (DATE 2012)*. IEEE, 983–987. https://doi.org/10.1109/date.2012.6176639.

[3] Thomas Carle, Manel Djemal, Dumitru Potop-Butucaru, Robert de Simone, and Zhen Zhang. 2014. Static Mapping of Real-Time Applications onto Massively Parallel Processor Arrays. In *2014 14th International Conference on Application of Concurrency to System Design*. IEEE. https://doi.org/10.1109/acsd.2014.19.

[4] G. J. Chaitin. 1982. Register allocation & spilling via graph coloring. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction - SIGPLAN'82*. ACM Press, 98–105. https://doi.org/10.1145/800230.806984.

[5] Vishwanathan Chandru and Frank Mueller. 2016. Reducing NoC and Memory Contention for Manycores. In *Architecture of Computing Systems – ARCS 2016*. Springer International Publishing, 293–305. https://doi.org/10.1007/978-3-319-30695-7_22.

[6] Jeonghun Cho, Yunheung Paek, and David Whalley. 2002. Efficient register and memory assignment for non-orthogonal architectures via graph coloring and MST algorithms. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems Software and Compilers for Embedded Systems (LCTES/SCOPES'02)*. ACM Press, New York, NY, USA, 130–138. https://doi.org/10.1145/513829.513853.

[7] Francesco Conti. CConvNet open source project. Retrieved July 10, 2017 from https://micrel-web-services.dei.unibo.it/brain-inspired/cconvnet-release.

[8] Francesco Conti, Antonio Pullini, and Luca Benini. 2014. Brain-Inspired Classroom Occupancy Monitoring on a Low-Power Mobile Platform. In *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops*. IEEE. https://doi.org/10.1109/cvprw.2014.95.

[9] Francesco Conti, Davide Rossi, Antonio Pullini, Igor Loi, and Luca Benini. 2015. PULP: A Ultra-Low Power Parallel Accelerator for Energy-Efficient and Flexible Embedded Vision. *Journal of Signal Processing Systems* 84, 3 https://doi.org/10.1007/s11265-015-1070-9.

[10] Benoit Dupont de Dinechin, Duco van Amstel, Marc Poulhies, and Guillaume Lager. 2014. Time-critical computing on a single-chip massively parallel processor. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014*. IEEE Conference Publications, 1–6. https://doi.org/10.7873/date.2014.110.

[11] S. Gautham and Erik Rainey. 2014. The Khronos OpenVX™ 1.0 Specification. https://www.khronos.org/openvx/.

[12] Georgia Giannopoulou, Nikolay Stoimenov, Pengcheng Huang, and Lothar Thiele. 2014. Mapping mixed-criticality applications on multi-core architectures. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014*. IEEE Conference Publications, 98:1–98:6. https://doi.org/10.7873/date2014.111.

[13] Andrés Goens, Jeronimo Castrillon, Maximilian Odendahl, and Rainer Leupers. 2016. An optimal allocation of memory buffers for complex multicore platforms. *Journal of Systems Architecture* 66–67. https://doi.org/10.1016/j.sysarc.2016.05.002.

[14] Min Kyu Jeong, Doe Hyun Yoon, Dam Sunwoo, Mike Sullivan, Ikhwan Lee, and Mattan Erez. 2012. Balancing DRAM locality and parallelism in shared memory CMP systems. In *IEEE International Symposium on High-Performance Comp Architecture*. IEEE, 1–12. DOI: https://doi.org/10.1109/hpca.2012.6168944

[15] Taewhan Kim and Jungeun Kim. 2007. Integration of Code Scheduling, Memory Allocation, and Array Binding for Memory-Access Optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26, 1 (jan 2007), 142–151. https://doi.org/10.1109/tcad.2006.882639.

[16] Yongjoo Kim, Jongeun Lee, Aviral Shrivastava, and Yunheung Paek. 2010. Operation and data mapping for CGRAs with multi-bank memory. *ACM SIGPLAN Notices* 45, 4 https://doi.org/10.1145/1755951.1755892.

[17] Ming-Yung Ko and Shuvra S. Bhattacharyya. 2003. *Partitioning for DSP Software Synthesis*. Springer Berlin Heidelberg, Berlin, Heidelberg, 344–358. https://doi.org/10.1007/978-3-540-39920-9_24.

[18] R. Leupers and D. Kotte. 2001. Variable partitioning for dual memory bank DSPs. In *2001 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (Cat. No.01CH37221)*, Vol. 2. IEEE, 1121–1124 vol. 2. https://doi.org/10.1109/icassp.2001.941118.

[19] Lei Liu, Zehan Cui, Mingjie Xing, Yungang Bao, Mingyu Chen, and Chengyong Wu. 2012. A software memory partition approach for eliminating bank-level interference in multicore systems. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques - PACT'12*. ACM Press, 367–376. https://doi.org/10.1145/2370816.2370869.

[20] Wei Mi, Xiaobing Feng, Jingling Xue, and Yaocang Jia. 2010. Software-Hardware Cooperative DRAM Bank Partitioning for Chip Multiprocessors. In *Network and Parallel Computing*. LNCS, Vol. 6289. Springer Berlin Heidelberg, 329–343. https://doi.org/10.1007/978-3-642-15672-4_28.

[21] Alastair Murray and Björn Franke. 2008. Fast source-level data assignment to dual memory banks. In *Proceedings of the 11th international workshop on Software & compilers for embedded systems - SCOPES'08*. ACM Press, 43–52. https://doi.org/10.1145/1361096.1361105.

[22] Vincent Nélis, Patrick Meumeu Yomsi, and Luis Miguel Pinho. 2016. The variability of application execution times on a multi-core platform. In *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*. http://www.cister.isep.ipp.pt/docs/the_variability_of_application_execution_times_on_a_multi_core_platform/1224/attach.pdf.

[23] Andreas Olofsson, Roman Trogan, Oleg Raikhman, and Lexington Adapteva. 2011. A 1024-core 70 GFLOP/W floating point manycore microprocessor. In *Poster on 15th Workshop on High Performance Embedded Computing (HPEC 2011)*. http://www.adapteva.com/wp-content/uploads/2011/10/adapteva_hpec11.pdf.

[24] Xing Pan, Yasaswini Jyothi Gownivaripalli, and Frank Mueller. 2016. TintMalloc: Reducing Memory Access Divergence via Controller-Aware Coloring. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 363–372. https://doi.org/10.1109/ipdps.2016.26.

[25] Quentin Perret, Pascal Maurere, Eric Noulard, Claire Pagetti, Pascal Sainrat, and Benoit Triquet. 2016. Temporal Isolation of Hard Real-Time Applications on Many-Core Processors. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 1–11. https://doi.org/10.1109/rtas.2016.7461363.

[26] B. Ramakrishna Rau. 1991. Pseudo-randomly interleaved memory. *ACM SIGARCH Computer Architecture News* 19, 3 https://doi.org/10.1145/115953.115961.

[27] Jan Reineke, Isaac Liu, Hiren D. Patel, Sungjun Kim, and Edward A. Lee. 2011. PRET DRAM controller: bank privatization for predictability and temporal isolation. In *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis - CODES+ISSS'11*. ACM Press, 99–108. https://doi.org/10.1145/2039370.2039388.

[28] Hamza Rihani, Matthieu Moy, Claire Maiza, Robert I. Davis, and Sebastian Altmeyer. 2016. Response Time Analysis of Synchronous Data Flow Programs on a Many-Core Processor. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems - RTNS'16*. ACM Press. https://doi.org/10.1145/2997465.2997472.

[29] Mazen A. R. Saghir, Paul Chow, and Corinna G. Lee. 1996. Exploiting dual data-memory banks in digital signal processors. *ACM SIGOPS Operating Systems Review* 30, 5 https://doi.org/10.1145/248208.237193.

[30] K. Shyam and R. Govindarajan. 2007. *An Array Allocation Scheme for Energy Reduction in Partitioned Memory Architectures*. Springer Berlin Heidelberg, Berlin, Heidelberg, 32–47. https://doi.org/10.1007/978-3-540-71229-9_3.

[31] Viera Sipkova. 2003. *Efficient Variable Allocation to Dual Memory Banks of DSPs*. Springer Berlin Heidelberg, Berlin, Heidelberg, 359–372. https://doi.org/10.1007/978-3-540-39920-9_25.

[32] Maria Soto, Marc Sevaux, André Rossi, and Johann Laurent. 2013. *Memory Allocation Problems in Embedded Systems: Optimization Methods*. Wiley-ISTE. 256 pages. https://hal.archives-ouvertes.fr/hal-00767031.

[33] StreamIt Benchmark Suite. Retrieved July 10, 2017 from http://groups.csail.mit.edu/cag/streamit/shtml/benchmarks.shtml.

[34] Andreas Tretter, Pratyush Kumar, and Lothar Thiele. 2015. Interleaved Multi-Bank Scratchpad Memories: A Probabilistic Description of Access Conflicts. In *Proceedings of the 52nd Annual Design Automation Conference on - DAC'15*. ACM Press. https://doi.org/10.1145/2744769.2744861.

[35] Andreas Tretter, Harshavardhan Pandit, Pratyush Kumar, and Lothar Thiele. 2014. Deterministic memory sharing in Kahn process networks: Ultrasound imaging as a case study. In *2014 IEEE 12th Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*. IEEE. https://doi.org/10.1109/estimedia.2014.6962348.

[36] Prathap Kumar Valsan and Heechul Yun. 2015. MEDUSA: A Predictable and High-Performance DRAM Controller for Multicore Based Embedded Systems. In *2015 IEEE 3rd International Conference on Cyber-Physical Systems, Networks, and Applications*. IEEE, 86–93. https://doi.org/10.1109/cpsna.2015.24.

[37] Zheng Pei Wu, Yogen Krish, and Rodolfo Pellizzoni. 2013. Worst Case Analysis of DRAM Latency in Multi-requestor Systems. In *2013 IEEE 34th Real-Time Systems Symposium*. IEEE, 372–383. https://doi.org/10.1109/rtss.2013.44.

[38] Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. 2014. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 155–166. https://doi.org/10.1109/rtas.2014.6925999.

[39] Lei Zhang, Meikang Qiu, Edwin H.-M. Sha, and Qingfeng Zhuge. 2011. Variable assignment and instruction scheduling for processor with multi-module memory. *Microprocessors and Microsystems* 35, 3 https://doi.org/10.1016/j.micpro.2010.12.002.

Received April 2017; revised June 2017; accepted June 2017