Circuits and Systems
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ens.ewi.tudelft.nl/

CAS-MS-2014-06

M.Sc. Thesis

Cache Balancer: A communication latency and utilization aware resource manager

Jurriën de Klerk B.Sc.

Abstract

The increasing number of processors in today's many-core architectures has lead to new issues regarding memory management. The performance of many-core processors is often limited by the communication latency incurred in data transfers between different cores. Conventional memory allocators do not take such communication costs into account while allocating memory for application tasks at runtime. While a number of existing proposals address this issue, they result in the non-uniform utilization of available system resources. This work introduces *Cache Balancer*, a technique for dynamic memory allocation that addresses the limitations of state-of-the-art schemes. Cache Balancer introduces the access rate metric to measure the utilization of different cache banks in the system, and uses this at runtime to determine where memory is allocated. The technique reduces memory access latency by up to 63.4% by avoiding allocation of memory in over-utilized cache banks. Furthermore, Cache Balancer incorporates a runtime task mapper that utilizes information on the execution characteristics of tasks and the structure of the system interconnect in determining a mapping solution that results in optimal memory throughput. This results in additional memory access latency reductions of up to 14.5%, and combined execution time improvements of up to 22% as compared to state-of-the-art schemes.

Faculty of Electrical Engineering, Mathematics and Computer Science

Delft University of Technology

Cache Balancer: A communication latency and utilization aware resource manager

Thesis

submitted in partial fulfillment of the
requirements for the degree of

Master of Science

in

Embedded Systems

by

Jurriën de Klerk B.Sc.
born in Amsterdam, Netherlands

This work was performed in:

Circuits and Systems Group
Department of Microelectronics & Computer Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Delft University of Technology
Department of
Microelectronics & Computer Engineering

The undersigned hereby certify that they have read and recommend to the Faculty of Electrical Engineering, Mathematics and Computer Science for acceptance a thesis entitled **"Cache Balancer: A communication latency and utilization aware resource manager"** by **Jurriën de Klerk B.Sc.** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: 22-09-2014

Acknowledgments

Over two years ago, before the work for this thesis project started, I asked dr.ir. René van Leuken for consultancy on possible thesis projects. When he pointed out one of the project as a 'high pain, high gain' project, I knew which project it should be. Usually I do things the hard way, since then you have to face more interesting challenges.

After two years of hard work I do understand what 'high pain' meant. The project was supposed to be a 9 months project. Throughout the project I faced many setbacks, which usually resulted in a postponed deadline. However I found the work itself extremely fascinating which helped me to overcome the many setbacks. Throughout the project I have gained an in deep understanding of many-core processors and data and task management, which helped me develop a passion for both the software and hardware for such many-core processors.

Therefore I would like to thank my first advisor, dr.ir. René van Leuken for his help with finding this project, but also for his assistance during the writing of this thesis. Also I would like to thank ir. Sumeet Kumar since he assisted me during the writing of this document as well. However I would like to thank him as well for the technical support he provided me. During this project we have had many discussions on different topic related to this work. Through these discussions I have learned a lot in both technical skills and the skills to convince people of my beliefs.

Jurriën de Klerk B.Sc.
Delft, The Netherlands
22-09-2014

Contents

List of Figures

List of Tables

1 Introduction

Over the past decades decreased transistor sizes have allowed system developers to place more functional blocks in a chip without increasing the used area. This development allows the integration of multiple *processing element*s (PEs) in one single chip, the basic concept for many-core processors. To increase the utilization of such a many-core processor, applications are partitioned into a set of tasks that can be executed concurrently by the different PEs. Because data used within an application might be shared by multiple tasks, the PEs are connected to a shared memory.

Off-chip memory access has a high latency compared to on-chip memory access, a problem that already existed with single-core processors. Therefore, on-chip memory, called a cache, is used to reduce the number of off-chip memory access attempts. Data, likely to be reused by the PE, is stored in the cache, which reduces the dependence on the off-chip memory. In many-core processors there are multiple PEs that can access memory.

The first challenge of many-core processors is how to allow PEs to access the memory simultaneously. Although caches reduce the average memory access latency, they are still serving only one memory access at the time. Therefore, if multiple PEs try to access the memory simultaneously, the different requests are still sequentialized. To overcome this limitation the single cache can be changed into several smaller caches. If multiple caches are used and data is replicated in multiple caches, a coherency issue can arise. If some value is replicated in multiple caches and the value is changed in one of them, the values in the other caches should be updated as well. Preventing any data replication avoids this coherency issue. A scheme that prevents data replication is partitioning the memory space into equal sections called memory pages. The memory pages are assigned to the different caches in a uniform manner. The content of a memory page is only allowed to be stored in the cache it was assigned to.

The second challenge that arises with many-core processors concerns the interconnect structure, connecting the caches and PEs. The interconnect structure should allow the different PEs to communicate with the caches simultaneously. Also the structure should allow to connect a large number of PEs and caches, thus the structure should scale well. For this reason, a *network on chip* (NoC) is a commonly used interconnect structure. A NoC connects multiple nodes to each other using point to point connections. Every node is connected to a limited number of other nodes. If communication is required between nodes who are not directly connected, information is routed through the NoC via one or multiple other nodes. The distance that information needs to travel from source to destination is measured in hops, where one hop is the distance between two nodes that are directly connected.

A result of the multiple caches and a NoC as interconnect structure is that the distance from a PE to the caches differs. Because every hop in a communication path introduces at least some latency, the latency to access a cache differs as well. Hence mapping data to a cache close to the processor reduces average memory access latency.

1.1 Motivation

In many applications the size of the required memory is not known in advance. During run-time memory must be allocated. To enable the dynamic behavior in applications to deal with the varying memory requirements, dynamic memory allocation is used. A memory allocator, which is responsible for dynamic memory allocation, provides applications with an unused memory section upon request. Conventional memory allocators look for the first free memory section. Therefore it is hard to predict to which cache the allocated memory maps to. It is likely that the distance from the PE requesting the memory section to the cache where the newly allocated memory maps to is not minimal.

A data management scheme that allocates memory in caches close to PEs was presented in [6]. This scheme allocates memory most of the time that maps to the cache bank closest to the requesting PE. However only using the caches that are close to the active PEs can limit the used cache capacity significantly. This limitation was addressed in the same research and solved by tracking the actively accessed memory pages per cache. If there are too many active pages stored in a cache, the distance aware scheme is more flexible allowing memory to be allocated that maps to a cache further away from the active PEs. The flexibilization of the distance aware scheme is simplified in [20]. In the simplified scheme, the difference in the number of allocated memory pages in the different caches is tracked in software.

A significant limitation in both [6] and [20] is that memory access patterns might be completely different for different memory sections. One memory section can be accessed many times, while other sections maybe used only once. Hence the utilization of the different cache banks can differ significantly. Because a cache bank processes memory requests one after the other, accessing a heavily utilized cache, results in a higher latency compared to a memory requests to a lightly utilized cache. Therefore, the first hypothesis is:

> *if the distance aware scheme is more flexible because the closest cache bank is heavily utilized, and as a result the allocated memory is mapped to a lightly utilized cache, it results in a reduced average memory access latency.*

If a task is mapped to a PE while distance aware memory allocation is applied, it could lead to the use of different caches compared to the situation where the same task was mapped to a different PE. Task mapping has influence on the location of the data. If two tasks share data then the access latency is reduced if the tasks are mapped to PEs close to each other. On the other hand mapping the tasks to two PEs close to each other could result in two PEs sharing a cache because of the distance aware scheme.

Then, in case the amount of shared data is very little the cache could become heavily utilized while other caches would still be idle. Hence, the second hypotheses is:

> *if a task mapping algorithm uses information about the communication latency due to memory access to the caches, the cache utilization, and used cache capacity, it can map tasks to PEs resulting in a reduction of memory access latency compared to conventional task mapping algorithms.*

This work theorizes that if cache utilization is considered in addition to the communication latency and used cache capacity during memory allocation, the memory access latency can be reduced. Furthermore, a task mapper that considers these same effects while tasks are mapped to the PEs can further improve the memory access latency. This work proposes a system to measure the cache utilization. This is integrated in a memory allocation scheme that reduces the memory access latency. In addition, a task mapping algorithm is proposed that uses information of the memory access characteristics of the tasks themselves to predict performance degradation due to communication latency and cache utilization. This prediction is used in a task mapping heuristic that improves memory access latency of the tasks.

1.2 Thesis goals

The main objective of this work is to develop a run-time manager that reduces memory access latency by allocating memory and mapping tasks both based on the distance information needs to travel while keeping a balanced utilization of the memory resources. In addition, a virtual platform should be developed to evaluate the run-time manager. Hence, the goals for this thesis are the following:

- The development of a memory allocation scheme capable of mapping data to the closest cache to obtain minimal memory access latency and to balance cache utilization.
- The development of a management scheme capable of mapping tasks, resulting in minimal performance degradation due to memory access latency.
- The development of a virtual platform capable of executing parallel workloads.
- The development of a run-time library providing support for applications executed on the virtual platform.

1.3 Contributions

The main contributions of this thesis are:

- The Cache Balancer: a memory allocation scheme that maps data to cache banks while making the trade off between communication cost, used cache capacity and cache utilization to reduce memory access latency, which is capable of reducing memory access latency up to 63.4%.
- Pain driven task mapping: a task mapping heuristic that predicts performance degradation caused by other tasks and uses this prediction to find a task map. The proposed scheme reduces the execution latency up to 14.5%.
- TMFab (version 2.3): a virtual platform simulating a scalable many-core processor capable of executing parallel workloads.

1.4 Thesis organization

This work is organized as follow: Chapter 2 explains the target architecture for the Cache Balancer. Also the concepts are explained which are used for the memory allocation scheme and the task mapper. Chapter 3 discusses the concepts of the memory allocation scheme and the task mapper. Then, Chapter 4 presents the virtual platform that is build to test out and verify the concepts presented in Chapter 3. The results of the verification are presented in Chapter 5 and the final chapter, Chapter 6, concludes this work.

Background 2

This chapter introduces concepts that are used in this work and based on existing technologies. This introduction begins with the basic principles of the architecture used in this work. Thereafter, transactional memory is introduced, which is used in the memory hierarchy of the many-core processor. Then, memory allocation is briefly explained and, finally, this chapter ends with a discussion on some of the existing strategies for task and data management.

2.1 A tiled based many-core architecture

A commonly used interconnect structure in many-core processors[1] is a *network on chip* (NoC), thanks to its scalability. The overall chip area is divided into tiles and each tile includes a router. The routers of all adjacent tiles are connected using a point-to-point connection, which builds the NoC.

A *processing element* (PE) tile contains, next to a PE, a private instruction cache and a *level 1 data cache* (L1\$) to limit the communication in the NoC. Most of these many-core processors use the *level 2 data cache* (L2\$) as a shared cache layer. To increase concurrency in the L2\$ layer, it is commonly implemented in several banks, allowing the PEs to access the different banks simultaneously. Depending on the design style of the many-core processor, the L2\$ banks are either included in the PE tiles or a separate tile is used for the cache banks [9][17]. Figure 2.1 depicts an example of a many-core processor that uses a separate tile for the different L2\$ banks.

Both the private L1\$ and the shared L2\$ is implemented in multiple separate caches. If data replication in the different caches within the same layer is allowed, a coherency scheme is required to keep the data inside these caches consistent. An approach to avoid a coherency scheme for shared caches is to deny data replication in the different caches within a cache layer. This can be achieved by dividing the address space into equal sections, and each section assigned uniformly to the cache banks. Thus each cache bank stores a unique subset of the address space. This technique introduces very little overhead and is completely independent of the system size. Moreover, it uses the cache capacity effective, since every unique data address maps to a unique cache set in the shared cache space.

Spreading the address space over the different L2\$ banks solves the coherency issue for the shared cache layer. Applying the same concept to the L1\$ results in a large increase in communication compared to a coherency scheme, since every memory reference to

[1]In this work many-core processor refers to a processor containing at least 8 processing elements.

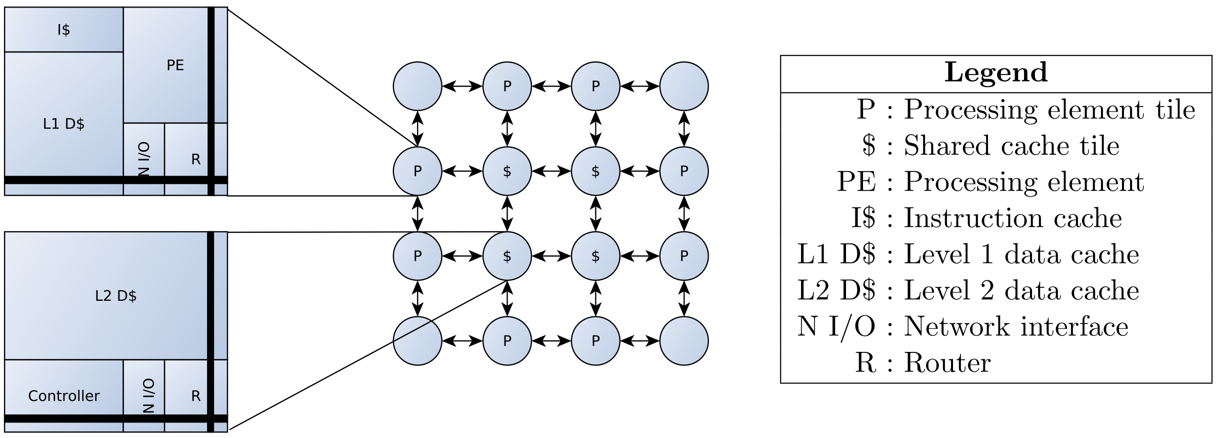


Figure 2.1: An example of a NoC structure.

an address that does not map to the local L1$ is served in another tile. A second way to avoid a cache coherency scheme is the use of transactional memories [17].

2.2 Transactional memory

One of the major problems of today's many-core architectures is the programmability. Adding more PEs to a system gives a potential performance gain, but it becomes more difficult to use all PEs efficiently at the same time. To use the different PEs efficiently, applications require a partitioning scheme, such that the sections, called threads or tasks, can be executed simultaneously.

If data is shared among tasks, it can result in one task modifying the shared data while another task is reading it. Ideally tasks do not share data, however in some cases this is unavoidable. To avoid unpredictable behavior, memory operation on shared data are executed in critical sections. Critical sections use mutual exclusion, forcing tasks to stall, in case some other task is executing a corresponding critical section. Placing the critical sections is mainly the issue in parallel application development.

The idea of transactional memory is to execute the critical section speculative. Hence a critical section does not force a processor to stall if an other processor is executing a corresponding critical section, but causes the first processor to ignore the second one, execute the section and check the validity of the performed work. The only reason work can be invalid is because conflicting data is used. This means that data was used in the critical section changed by another PE. There are two different methods for conflict detection. The first, called *optimistic conflict detection*, requires a validation of the used data at the end of the critical section. The second method validates data at every memory access and is, therefore, called *pessimistic conflict detection*. A transactional memory scheme that uses optimistic conflict detection to avoid a cache coherency scheme was presented in [17].

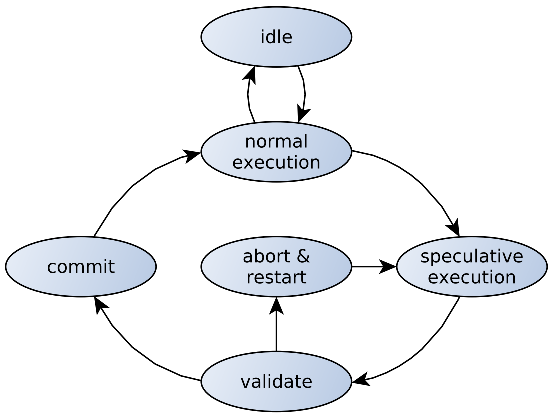


Figure 2.2: Basic execution flow for transactional memory.

Figure 2.2 depicts a state diagram that shows the execution model for a transactional memory scheme using optimistic conflict detection. At the beginning of a task, the processor enters the normal execution state. If the PE encounters a critical section, the processor executes the section in the speculative execution state. In this speculation state, the processor assumes that the data used during the execution of the transaction is not changed by any other PE. When the end of the transaction is reached, the processor enters a validation process where the PE checks whether this assumption was correct. In case the assumption holds, the PE commits the changes made during the transaction to the L2$. In case the assumption fails, the processor rolls back and restart the critical section. These speculative executed sections are called transactions.

2.3 Memory allocation

In many application the size of the required memory is not known until run-time, thus memory must be allocated when the application is executed. To enable the dynamic behavior in applications to deal with the varying memory requirements, dynamic memory allocation is used. A memory allocator, responsible for dynamic memory allocation, provides applications with an unused memory section of sufficient size upon request. A memory allocator guarantees either to return a pointer to a memory section that is not in use by any application, or informs the requester that there is no free memory left to allocate. The second guarantee provided by a memory allocator is that any memory released by an application is not lost and can be reused later on. This section discusses briefly the concepts of memory allocation, as this is an important tool to manage data in many-core processors.

An application defines three different sections within the memory space: instructions, heap, and stack. The instructions, which define the applications behavior, are generated by a compiler and placed in the instruction section. Most variables used as local variables inside functions are stored in the stack space. Global variables visible in the

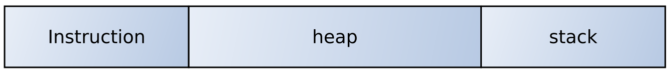


Figure 2.3: Default GNU memory map.

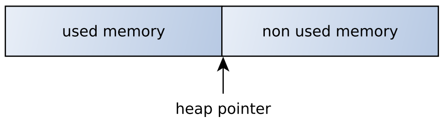


Figure 2.4: The heap section.

entire application and the local variables not placed on the stack are stored in the heap section. Figure 2.3 depicts a layout as the GNU compiler generates by default.

A memory allocator uses the heap to allocate memory. The heap pointer points to the edge between used and non used memory, as shown in Figure 2.4. When memory is allocated, the heap pointer is shifted into the nonused memory, thus leaving a nonused section in the used memory area. This section is then returned as the new allocated memory section.

The discussed memory allocation scheme works well when there is just one PE. However, when there are multiple PEs that might try to allocate memory simultaneously, this could result in a lot of conflicts, as they all try to modify the same heap pointer. The use of mutual exclusion or transactions results in a working system but would sequentialize all memory allocations.

To improve the memory allocation process for many-core processors, the allocator can be divided into a front- and back-end. The back-end provides the front-end with large memory sections, using the heap pointer, when required. The front-end uses these large memory sections as a small heap for memory allocation. Because the back-end can provide multiple of these small heaps, the front-end can use a heap per PE. When a PE requests a memory section, the request is served by the front-end using such a small heap reserved for the requesting PE. Therefore, if multiple PEs request memory sections simultaneously, they can be served in parallel. The heap pointer is only accessed when a per PE heap does not include a memory section of sufficient size [2].

2.4 Data and task management

This section discusses previous work on data and task management.

In 2006, Cho & Jin [6] presented a memory allocation scheme implementing *Distance awareness*. They presented a concept where the memory page allocation scheme is aware of which processor is requesting the memory page, and which cache bank the page maps to. Their scheme allocates memory pages mapped to the cache bank closest to the requesting PE. Cache capacity misses due to the allocation of too many memory pages that map to a single cache bank, called *Cache overflow*, was addressed in this same research as well. They avoided this effect by tracking the actively accessed memory pages per cache bank in hardware. If the number of actively accessed memory pages reached a certain threshold, pages that map to a cache bank one hop further away where allocated instead. The latter was simplified in [20], where software counters where used to track the difference in the number of allocated memory pages per cache bank. However, both [6] and [20] neglected cache utilization in their memory allocation scheme.

In 2009, Hardavellas et al. [10] investigated the different data types in a shared memory architecture. They applied this knowledge to determine which cache bank should store what data in a run-time manager called Reactive-NUCA. The scheme requires a cache coherence system to keep data in the L2$ banks consistent. They did not consider cache overflow or cache utilization in their scheme. In 2009, Jin & Cho [12] did research on cache access patterns. They used a tool to generate hints at compile time, which are used in a software run-time system, called SOS, to decide which data should map to which cache bank. However, their study only focused on the access patterns and communication latency and did not include any knowledge on the used cache capacity and neglected the impact of cache utilization.

A study on the difference of *Cache overflow* and *Hot caches*, referring to over utilized caches, was presented by Tang et al. in 2011 [21]. They showed that this has not the same effect, and that the use of last level cache misses as prediction on the hotness of a cache can be misleading. In their work, they present an application scheduling algorithm based on these principles and on-line measurements. The scheme was constructed for an architecture using a tree based cache hierarchy. Hence, distance awareness was not considered. Moreover the PEs could only cause one cache to become hot, and the number of PEs that could cause a cache to be hot was limited to two. Hence, the miss rate in the private caches can be compared to determine the hotness of a shared cache.

If distance awareness is applied, memory to be allocated maps to a cache bank close to the PE that is using the memory. Which cache bank is the closest depends on which PE is executing the task. Task mapping has a significant impact on memory allocation when distance awareness is used. If data is already stored in a cache bank, then mapping a task to a PE far away from that cache bank results in a higher communication latency compared to a PE close to the cache bank, as it was shown in [1]. However their research did not consider cache access characteristics. A different approach to reduce communication latency was presented in [7]. This research minimizes the number of PEs that share a link using task mapping. However, like [1], cache access characteristics are neglected.

Majo & Gross presented in 2011 [16] research that focused on non uniform memory architectures. Their architecture uses a tree based cache hierarchy and two *external random accessible memory*s (XRAMs). The architecture included four PEs, which are grouped in pairs of two PEs, and each par is connected via two cache levels to a XRAM. In their work they present a scheme for thread scheduling based on profiling. However, the scheme only considers PEs to use memory directly connected to a XRAM or indirectly via the other PEs cluster. Also they combine the effect of cache overflow and hot caches into a single measure.

A study on different application scheduling algorithms was presented by Zhuravlev et al. in 2010 [7]. They introduced two new application scheduling concepts. The first was based on the concept of *Pain*, which is a prediction of performance degradation caused by one application sharing resources with a second application. The second measure they used for application scheduling was the last level cache miss rate. However both schemes where developed for an architecture using a tree based cache hierarchy. Therefore, distance awareness was considered as irrelevant. Also the pain caused by an application was only considered for a single cache.

2.5 Conclusion

A common architecture for many-core processors is a tiled based structure, connection via NoC. The L2$ layer is implemented in multiple bank, spread over the system. A memory allocator can select the cache bank where the data maps to. Also task mapping has a significant impact in data mapping, if distance awareness is used. The next chapter discusses both memory allocation and task mapping in more detail.

Memory allocation and task mapping 3

As the shared cache layer is implemented in multiple banks and PEs can access all cache banks, there is the possibility that communication paths are longer than required. This can possibly be avoided by using data and task management. This chapter explains strategies to manage both data and task and discuss the different effects that might occur when implementing them.

3.1 Memory management

In this section strategies for memory management are explained. Section 3.1.1 shows the current state-of-the-art. Section 3.1.2 explains the memory allocator of *Cache Balancer*, a new concept introduced in this work.

3.1.1 State-of-the-art data management

This work is focusing on the type of many-core PEs, introduced in Section 2.1. If data is distributed in a round robin manner over the memory address space, then there is a potential that a PE on one side of the system uses memory that maps to a cache bank on the other side. This means that the length of a communication path, measured in the number of hops, can vary depending on which cache bank is used by a PE. Assuming that every hop in such a path introduces at least one cycle latency, a longer communication path results in a larger latency. Another aspect is that every link in any communication path can be used by another node pair. This could cause one communication path to block another. If the average hop-count of the communication paths increases, the chance that different communication paths share links is increasing as well. This interference of one path blocking another is increasing the latency even further. Similar, assuming that every hop increases the energy required for communication, a shorter communication path reduces energy consumption, as a result minimizing the hop-count in the communication paths reduces latency and energy consumption.

As explained, different memory sections map to different cache banks. As memory is allocated to store data and a memory allocator is free to choose which memory section to use, memory allocation can manipulate the bank selection for data. Allocating memory mapping to a cache bank close the requesting PE minimizes the length of the communication path. Figure 3.1 depicts a heat map showing the different hop-counts from the highlighted blue PE to the different cache banks.

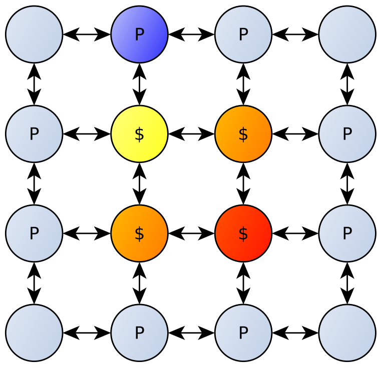


Figure 3.1: Distance from a PE to the different caches.

A scheme proposed in [6] achieves this minimization using the explained concept. To apply this concept, the memory space is divided into memory sections of equal size, where any consecutive section maps to a different cache bank. Thus, if the cache bank selection is defined by

$$C = \frac{Address \mod 2^k}{2^m} \tag{3.1}$$

where C is the cache bank that stores data with address $Address$, k and m define the bits used from the address to determine the cache bank, with $m < k$, then the section size is equal to 2^m. If a PE requires more memory, it allocates memory from a section that maps to a cache bank close to the PE .

This memory allocation scheme restricts the allocator to use only memory that maps to exactly one cache. This technique reduces the average hop-count in the communication paths between PEs and cache banks, but it also reduces the available cache capacity for a PE. In the situation where not all PEs are active, any cache bank with a certain distance to the active cores is unused. If the active cores use a large data set, larger than the cache capacity they are allowed to use due to distance awareness, then this results in cache misses in the shared cache banks. Also the costs of the use of a longer communication path is lower compared to the costs due to a cache miss. If the total cache capacity in the shared cache layer is utilized better, it results in a reduced average memory access latency and energy consumption. Hence, in case a cache suffers from cache overflow, referring to cache misses due to the limited capacity, the distance aware scheme should be less strict, allowing PEs to allocate memory mapping to cache banks further away and, therefore, utilizing the cache capacity better. The concept of making the distance aware scheme more flexible, allowing PE to allocate memory that maps further away, is called *memory page dispersion*.

If page dispersion is required, the distance between the requesting PE and the used cache bank is still important. An increased hop-count in the communication paths still has an increased latency. However the costs due to increased hop-count is lower compared to the costs due to a L2$ miss. Therefore, if the closest cache bank suffers from cache overflow, a memory section should be selected that maps to the next closest cache bank. If the next closest cache bank also suffers from cache overflow, this would still result in a L2$ miss. Therefore, not only the communication distance should be considered, but also if the selected memory maps to a cache bank that suffers from cache overflow.

One approach to this problem is counting the active accessed pages in a cache bank, as it was proposed in [6]. In that work, a dedicated hardware unit registers all pages that are accessed in a cache bank. The information of this unit shows how much cache space is used per cache bank and, thus, shows whether the cache bank suffers from cache overflow.

An alternative scheme that simplifies the detection of cache overflow was proposed in [20]. This proposal consists of an algorithm counting the difference in the number of memory pages allocated in the cache banks. This proposal showed that this algorithm can achieve similar performance as [6], but did not require any changes in the logic circuitry.

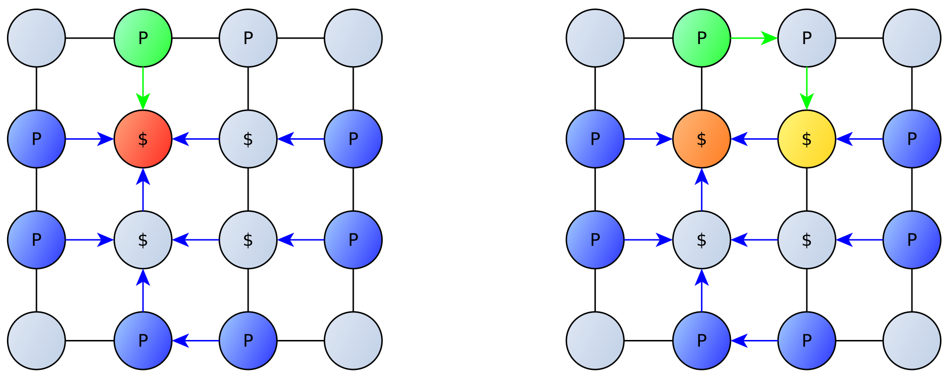
The algorithm proposed in [20] uses a counter for each shared cache bank. The counters are incremented when a memory page is allocated in the corresponding cache bank. When memory pages are allocated the algorithm compares the counter values against a threshold. If a counter value exceeds the threshold of the corresponding bank, it is avoided for memory allocation. To guarantee that at least one bank is allowed for memory allocation, the next two formulas are used as invariants by the algorithm.

$$\exists_i[c_i = 0] \tag{3.2}$$
$$\forall_i[c_i \geq 0] \tag{3.3}$$

where c_i is the counter value that corresponds to cache bank i. If the first invariant is not fulfilled after memory allocation, all counter values are equally decremented until both invariants hold. When memory is deallocated, the same action is performed in reverse order. Thus, decrementing the counter of the cache bank where the deallocated memory page maps to. If the second invariant does not hold, all counters are equally incremented until both invariance are fulfilled.

The value of the threshold can be used to influence the memory page dispersion degree. When the threshold is 0, the cache bank selection approaches a round robin scheme. When set to infinite, the scheme behaves like the distance aware scheme.

Both [6] and [20] did not consider the cache utilization in their work. As it was shown in [21], cache overflow and cache utilization are not the same effect, and should be considered separately when memory sections are selected for allocation. In [21], cache utilization was measured using the miss rate in the private caches. This works well if these misses are always served by the same L2$. However if the memory address of the miss is used to determine which cache bank handles the request, the private cache miss rate provides only an indication on the overall utilization of the L2$. The



(a) A hot cache (b) Hot cache avoidance

Figure 3.2: Avoiding over utilized caches.

next section discusses how cache utilization can be measured in a many-core processor, implementing the shared cache layer in multiple banks.

3.1.2 The Cache Balancer

This section introduces the *Cache Balancer*, a scheme improving state-of-the-art by taking a third important effect into consideration for memory allocation. This third effect impacting system performance is then cache utilization. When multiple PEs access one single cache bank, the requests are sequentialized, and it is slower compared to PEs accessing different cache banks.

Figure 3.2a depicts a situation where a large number of PEs share data in a certain cache. In this figure the blue PEs access the red cache for shared data. The green PE allocates private data in the same cache due to distance awareness. If the green PE allocates memory in a cache one hop further away, the access latency is reduced, since the request can be served directly, as shown in Figure 3.2b. A *hot cache* refers to a cache bank that suffers from over utilization. A *cold cache* is an under utilized cache bank.

The Cache Balancer improves the state-of-the-art by taking the hotness of caches into consideration. When a PE requests a memory section, the Cache Balancer detects, first, whether the cache bank closest to the PE is hot. If this is not the case, then cache overflow is considered. If the cache bank is neither hot nor overflowing, a memory section is allocated that maps to the closest cache bank. In case the cache bank suffers from either hotness or overflow, the Cache Balancer looks for a memory section that maps to the next closest cache bank, and repeats the process.

Cache overflow is detected using a similar algorithm to the one presented in [20]. Measuring whether a cache is hot could be achieved by counting the number of accesses to a given cache in a certain time frame. With the use of a threshold, the decision on whether a cache is considered to be hot could be straightforward. However, this could result in all cache being hot and, hence, the information whether caches are hot is lost.

To avoid this, the Cache Balancer uses a relative measure. The relative measure, called *access rate*, is defined by:

$$A_i = \frac{a_i}{a_t}$$

where A_i is the access rate for cache bank i, a_i is the number of accesses to cache bank i and a_t is the number of accesses to the overall shared cache layer. The access rate shows how heavily a cache bank is utilized with respect to the other cache banks. If a cache bank has an access rate of 1, then all other cache banks have a access rate of 0. This means that all the L1$ misses were served by the cache bank with an access rate of 1.

If the total number of cache accesses is low, the access rate shows that the caches that are accessed are hot, while this might not be the case. This could cause a PE to apply page dispersion while not necessary. To avoid this behavior the allocating PE needs more information than only the access rate. Combining the access rate with the number of accesses per bank could give good insight whether a cache bank is hot or cold.

The Cache Balancer uses a threshold to decide whether a cache bank is considered as hot. The threshold for the access rate can influence how quickly a cache bank is considered to be hot. A threshold of 1 almost ignores any hot caches, since 1 is the maximum value for the access rate. Thus, if more than one cache bank is accessed, all cache banks are considered cold. A threshold of 0 makes all cache banks hot.

If the difference between the access rate of two cache banks is small, the difference in utilization is small as well. If the cost do not outweigh the benefits of selecting a different cache bank due to utilization, then page dispersion should not be applied. Hence, there should be a significant difference between the access rates of the hot and cold cache. The result is that a cache bank can be warm. This means that memory that maps to such a cache bank can be selected because of distance awareness, or to avoid cache overflow, but is not used to avoid a hot cache.

To achieve this, two different thresholds are required: one for deciding whether a cache bank is hot, and one to decided whether a cache bank is cold. If the distance between a selected cache bank and the PE increases, the communication costs increase as well. The benefits of memory accesses to a lightly utilized cache bank should, at least, be larger than the communication costs. Thus, the threshold to decide whether the cache bank is cold should depend on the distance to the requesting PE. Therefore, the threshold functions are defined by:

$$\begin{aligned} t_{hot} &= \alpha \cdot midrange(A) \\ t_{cold} &= \tfrac{E[A]}{\beta \cdot HC(p_i, m_j)} \end{aligned}$$

where t_{hot} and t_{cold} are the thresholds, α and β are parameters that control the distribution of data in the cache banks, $HC(p_i, m_j)$ is the number of hops between PE i and cache bank j, and $midrange(A)$ is the midrange of the access rates:

$$midrange(A) = \frac{\max(A) + \min(A)}{2}$$

where A is the set of access rates. α influences the decision on whether a cache bank is considered to be hot and, thus, how heavily a cache is utilized before it is avoided by the memory allocator. The β value influences the data distribution. A low β value allows data to be spread further away from the PE, a high β value results in memory allocated closer to the PE.

In order to measure the access rate it is required to integrate an access counter per cache bank and a counter for the total number of accesses. A shared cache bank access increments the counter of the cache bank and the total access counter. Next to the counters, a number of registers are required to store the access rates. To make a proper prediction on the distribution of the access rate, a walking average is used. After a certain number of clock cycles the counters are sampled and averaged with a number of previous samples, creating the new access rates. To limit the required calculation units inside the cache banks the registers can store the counted values. This comes with the cost of an additional register per cache bank to store the total number of accesses and an additional flit in the communication packet for reading the counter values. Besides the limitation of the required calculation units, this also allows the PE to read the counter values to combine the access rate with the number of accesses avoiding page dispersion when all cache banks are cold.

```
Input: Initial cache bank ($init$), Number of memory pages ($npage$)
Result: Selected cache bank
i = init;
if $A_i > t_{hot}$ || $c_i > t_{overflow}$ then
    i = getNextCache();
    while $A_i > t_{cold}$ && $c_i > t_{overflow}$ do
        i = getNextCache();
        if $i == -1$ then
            i = init;
            break;
        end
    end
    if $c_i == 0$ then
        min = findMinPageCounter();
        for $j = 0$ To NUMBER_OF_CACHES do
            $c_j = c_j - min$;
        end
    end
    $c_i = c_i + npage$;
end
return $i$;
```

Algorithm 1: Cache bank selection.

Algorithm 1 shows the pseudo code for the cache bank selection. In this algorithm both cache overflow and hot caches are considered to find a cache bank. The input of the algorithm is the initial cache bank that was selected by the distance aware memory allocation and the number of memory pages to allocate. The output is the selected cache bank. The `getNextCache` function returns the next closest cache bank, or -1 in case all cache banks have been searched. The `findMinPageCounter` function returns the minimum value of the memory page counters used to avoid cache overflow.

3.2 Task mapping

In a shared memory architectures, there is a difference in shared and private data. Private data is used by a single PE. The memory management schemes can be applied while allocating private data. For shared data, which is data used by at least two PEs, the schemes might be less effective. In case tasks, who share data, are mapped to opposite sides of the system, it might result in poorly performance of the management schemes. Hence, good task mapping is important in the context of data management.

If the described data management schemes are used, then task mapping changes the overall memory allocation. If a task is mapped to a PE, the used data is allocated close to that PE. If the task was mapped to another PE, it can be the case that there is another cache bank closer and, thus, the data is allocated in a different cache bank. This only holds for the private data of the task, the shared data might still map to the same cache bank. Hence mapping tasks influences memory allocation, but with a different granularity as direct memory allocation does. Task mapping changes the location of the complete private data set used for the task by the PE in stead of separate memory sections.

The decisions about what task should be mapped to what PE, should have the same considerations as the memory management schemes. A cluster of cache banks can be selected to guide the task mapper to use a certain area of the many-core processor. A tasks should be mapped to a PE that is close to the shared data used for the task. However the amount of shared data used for a task might be very small or non at all. Thus the distance of the PE executing the task and the cache storing the shared data might be completely insignificant. In that case, cache overflow avoidance or hot cache avoidance might be the significant factor for task mapping.

The allocation of the shared memory can be something known at run-time, and could be used by a task mapper. However which task is going to use what data, what data is shared among tasks and what the ratio is between shared and non shared data is still unknown. Hence, deciding how important the different effects are is almost impossible without more knowledge about the overall behavior of the application and the tasks within applications. Before considering how this information could be obtained the information requirements should be known.

The next sections presents the concepts for task mapping used in this work. The section starts with a discussion on a distance aware based approach in Section 3.2.1, followed by the introduction of the pain measure in Section 3.2.2 and ends with the explanation of the scheme introduced in this work and the discussion on the validity of the assumptions in Section 3.2.3 and Section 3.2.4, respectively.

3.2.1 Distance aware task mapping

If the task mapper knows which task uses which data, then it can map the tasks as close to the shared data as possible. In case multiple tasks share data, it could be the case that not all tasks can be mapped with a single hop distance to the cache bank storing the shared data. Even if the task share the same data, one task can have higher memory throughput requirements for the shared data compared to another task. Thus, it matters which task is placed closer to the target cache bank. These requirements mainly depend on two variables, the L1$ miss rate of the PE executing the task and the amount of requested data for the task. To make the latter practical to use, it is defined by the load/store rate:

$$LS = \frac{memory\ operations}{instructions} \tag{3.4}$$

where *memory operations* is the number of memory operation instructions for a task and *instructions* the total number of instructions.

By using the load/store rate LS and the miss rate, a mapping strategy can be constructed, which minimizes communication latency. The tasks are ordered descending on the combination of both the load/store rate and the miss rate, then mapped one at the time to a free PE that is located as close as possible to the cache banks with the shared data.

Mapping tasks as close as possible to the cache banks where the shared data maps to could result in two PEs sharing a cache bank for their private data. If the tasks use more private data than shared data, it could be better not to map the tasks close to this cache bank, as this is not the bottleneck for the execution of the application. Spreading the tasks potentially increases communication latency, but when tasks use more private data as shared data, then this might avoid cache overflow or hot caches.

3.2.2 Pain; predicting performance degradation due to shared resources

Zhuravlev et al. presented in [24] different strategies to recognize whether applications could share memory resources without losing too much performance. In their research, they show that a scheme based on static analysis performed best, but that a similar scheme based on the last level cache miss rate showed similar quality. They reasoned that this was better applicable, since this required less changes in the compiler. However miss rate is something that can only be measured at runtime, moreover if tasks should be managed according miss rate then the management scheme can only start when the tasks are already running. This implies that tasks should migrate in case the miss rate in the last cache level becomes unbalanced. This could work well with the processor architecture they use. This architecture uses a tree based memory hierarchy. The cache coherency scheme in this type of memory hierarchy forces data migration if required, as PEs are not allowed to access far away caches.

In a tiled based many-core processor, the PEs are allowed to access all the different cache banks. Thus, this type of data migration does not occur. As a result, a scheme should decide which data should be migrated and when the migration should happen. The decision, as well as the migration, introduces a latency and, hence, it would be better if avoided.

The scheme based on static analysis proposed in [24] introduced a measure called *Pain*. Pain is a combination of two different parameters, cache sensitivity and cache intensity. Cache sensitivity indicates how much an application suffers when cache space is taken away from it by another application sharing memory resources. Cache intensity estimates how aggressively an application is using the cache. The cache intensity is predicted using the number of memory operations. Cache sensitivity is defined by

$$S = (\frac{1}{n+1}) \sum_{i=0}^{n} i \cdot h(i)$$

for a n set associativity cache and where $h(i)$ is a function of the reuse distance of the application.

If application A and B are mapped, in such a way that they share a cache, then the pain for application A caused by application B is

$$Pain(A_B) = S(A) \cdot Z(B)$$

where $S(A)$ is the sensitivity of application A, and $Z(B)$ is the intensity of application B. The total pain caused by the applications sharing one cache is

$$Pain(A, B) = Pain(A_B) + Pain(B_A)$$

The pain measure is used to decide whether two applications can be scheduled in such a way that either they share a cache, or it is better not to schedule the application as such.

Analyzing the principles of this scheme reveals that Pain is a measure closely related to both cache overflow and hot caches. The sensitivity measure shows how likely it is that some application causes cache to overflow. Cache intensity shows how likely it is that an application causes a cache to become hot.

The scheme based on the Pain measure was proposed to apply on applications scheduled on a multi-core PEs using a tree based memory hierarchy. Section 3.2.3 explains a new concept introduced in this work based on the concept of Pain.

3.2.3 Pain prediction in a tiled based architecture

In many-core processors the PEs can access all shared cache banks. Tasks could share data possibly stored in multiple caches. In other words, one task could cause pain for several other tasks. Mapping a task to a PE fixes the communication paths from that PE to the cache banks storing the data for the task. The task mapper should be aware of this and take it in consideration while mapping tasks to PEs. Therefore, the task intensity is defined by:

$$Z(m_i) = \sum_{t_j \in T_i} Z_i(t_j, m_i)$$

where m_i is cache bank i, t_j is task j, T_i the set of tasks that is using m_i and $Z_i(t_j, m_i)$ is the intensity per cache caused by t_j in cache m_i. Therefore:

$$Z(A) = \sum_{m_i \in M} Z(m_i)$$

where M is the set of cache banks and $Z(A)$ is the applications cache intensity. The $Z_i(t_j, m_i)$ is estimated by the number of memory operation in t_i that access data stored in m_i. Using the per cache intensity pain can be estimated as task pain

$$Pain(t_i) = \sum_{m_j \in M_{t_i}} S(t_i) \cdot Z(m_j)$$

where M_{t_i} is the set of caches used by t_i, and $S(t_i)$ equal to the original $S(A)$ only now the sensitivity is narrowed down to t_i instead of the sensitivity of the entire application. These functions are closely related to the original functions but at a finer granularity. This is required for a task mapper designed for a many-core processor.

To integrate distance awareness into the task mapper communication latency due to an increased hop-count should cause pain as well. What the hop-count is in the different communication paths from the PE executing the task to the cache banks storing the data, depends on which PE was selected by the task mapper. Hence, the communication pain makes use of the temporal mappings function

$$map(t_i) = p_q$$

where p_q is a free PE. The temporal mappings function is only used to evaluate the pain for a task when it is mapped to a PE. By using the mappings function, a communication pain prediction is defined by

$$CP(t_i \mid map(t_i) = p_q) = \sum_{m_j \in M_i} 2 \cdot HC(p_q, m_j) \cdot Z(m_j) \cdot S(t_i)$$

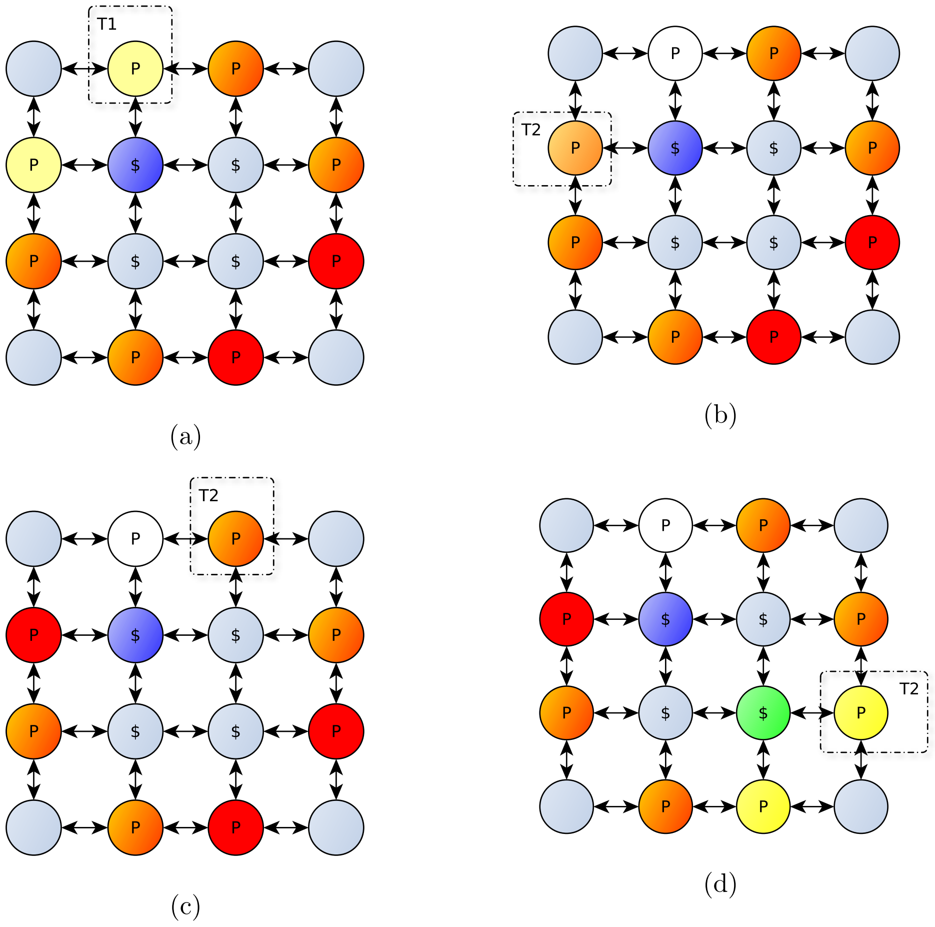


Figure 3.3: Pain due to the combination of cache utilization and communication.

where the function $HC(p_q, m_j)$ is the hop-count of the communication path from p_q to m_j. This function is doubled since every memory access consists of a request and a response. Then, finally the pain mappings function can be defined by

$$pmap(t_i) = \min_{p_j \in P}(Pain(t_i) + CP(t_i \mid map(t_i) = p_j))$$

which is used as mathematical description of the task mappings function used by the mapper introduced in this work.

Because the mapping scheme is applied at run-time, an algorithm should be used which avoids the search through all possible mappings. The same algorithm, as explained in Section 3.2.1, can be applied, with the difference that for task assignment the *pmap* function is used to decide whether a task maps to a PE.

Figure 3.3 shows four different examples of pain distribution in a many-core processor. Figure 3.3a shows the situation where a task is using data in the blue highlighted cache. As no other PE is using this cache, and since the communication pain for the PEs close to the blue cache is low compared to the other PEs, these PEs introduce very little pain.

If a second task is mapped, using the same shared data, a possible pain distribution is depicted in Figure 3.3b. This figure assumes that the shared data is larger compared to the private data. If this is not the case, Figure 3.3c could be the result. The pain distribution is depicted in Figure 3.3d in case the second task uses different shared data.

In the worst case scenario, the sorting algorithm is $O(n^2)$ but on average $O(n \cdot \log(n))$ for n tasks [11]. To find the PE to map a task to is $O(p)$, where p is the number of PEs allowed to execute the tasks. Thus, the overall algorithm is $O(n \cdot p \cdot \log(n))$ on average and $O(p \cdot n^2)$ in the worst case.

3.2.4 Gathering hints for task mapping

The next list sums up the used parameters as reminder of what was required by the mapping algorithm:

P	the set of free PEs that are allowed to execute one of the tasks
$HC(p_i, m_j)$	the hop-count in the communication path from PE i to cache j
LS	the per task load/store rate
$Z(t_i, m_j)$	the per cache intensity in cache j caused by task i
$S(t_i)$	the cache sensitivity for task i
MR	the per task L1$ miss rate

Which PE is free and which PE is executing a task is known at run-time. To limit the search through the free PEs, a scheduling unit can select a subset of the free PEs and allow only the selected PEs to execute the tasks. The selection of such a subset is mainly important when a scheduler is involved with knowledge of possible other executed applications. The algorithm involved in the subset selection is kept out of scope in this work and any free PE is allowed to execute one of the tasks. Hence, P, the set of free PEs allowed to execute a task, is known by the task mapper.

The hop-count function $HC(p_i, m_j)$ depends only on the topology of the many-core processor. Once this is implemented in silicon, the topology is fixed and can be stored as constant look-up table in memory. This function is also required for any distance awareness. Hence, it does not introduce any additional costs.

In [5], it was shown that it is possible to track down a load/store rate. However, in this work it was assumed that every loop was iterated just once. This one loop iteration load/store rate is not equal to the one defined in Equation 3.4. However, it could still serve as a rough estimation. If it is possible to predict this more accuracy, it would be preferable.

If the number of iterations in a loop is dependent on a constant, then the number of memory operations and the number of instructions within the loop could be multiplied by this constant and result in an exact number. This is not possible when the number of iterations does not depends on a constant. In most cases the compiler does know on which variable the number of iterations depends. If the compiler can trace the variables back to the point of task mapping, a hint can be injected. This hint can be evaluated at run-time to determine the load/store rate for the used input data set. As the cache intensity is defined as the total number of memory operations, the compiler can predict this number in the same way. In case the compiler is not capable of providing an accurate hint, the rough estimation can be used.

The per task reuse distance could be generated using profiling. The downside of profiling is that this gives a number for a specific data input set. Ding & Zhong in [8] showed that it is possible to combine two profiling runs and pattern recognition to produce an parameterized model for the reuse distance that only depends on the size of the input data. The latter is, in most cases, known when task mapping takes place. The parameterized model for the reuse distance can be injected as hint by the compiler. Finally Zhong et al. showed in [23] that, using the parameterized model generated with the techniques presented in [8], a parameterized model can be generated for the miss rate. Thus this can be injected as hint as well.

The techniques used to generate the parameterized models for both the reuse distance and the miss rate only works well when the application produces predictable access patterns. Even though this is often the case, it can happen that these predictions have a significant error.

3.3 Conclusion

The distance aware scheme has big potential, especially when applied in a large system. This reduces the distance packets needs to travel. It decreases communication latency and NoC energy consumption, but this might cause cache overflow or hot caches.

Page dispersion might overcome the cache overflow problem introduced by the distance aware management scheme. However, on its own, it almost performs equally to a round robin bank selection scheme, since round robin bank selection avoids cache overflow as much as possible. If combined with distance awareness, this could increase the chance of hot caches, since the chance that multiple PEs are using data stored in a single cache bank is increased when page dispersion is applied.

Page dispersion due to hot caches could avoid hot caches, but on its own, it performs again close to the round robin scheme. Depending on the application, it might perform a little better since hot caches are avoided. Anyhow, if the access rates for the different cache banks do not differ significantly, then it is similar to a round robin scheme. In other words, it is likely that all three management schemes show big potential in theory, but might not perform in reality if only one is applied. Applying all the three concepts should overcome the different problems.

For the task mapping algorithm, static analyses and profiling are used to generate hints. These hints are used to predict the behavior of the task to map and evaluated at run-time to guide task mapping. The algorithm considers the same effects used in memory allocation. Therefore, at first glance, the data management scheme may look redundant. However, as explained in Section 3.2.4 the hints are not alway as accurate as they need to be to generate a good map for the tasks. In this case the memory allocation schemes are making up for possible mapping errors. Also some applications produce hot caches or cache overflow independent of the task map, as this is the behavior of the applications. Hence, both the task mapping algorithm and the memory allocation schemes should be used.

The next chapter presents the virtual platform that is used to develop and evaluate the concept introduced in this chapter. Also the run-time library is introduced in the next chapter. This library implements a memory allocator and a task mapper which use the concepts introduced in this chapter.

TMFab many-core architecture 4

This chapter describes the concepts of the *Transactional Memory Fabric* (TMFab) architecture, which forms the base platform used for this work. This chapter begins with a description of the components used within TMFab and, then, provides an overview of the transactional memory scheme used for lock-free accesses to shared data. Furthermore, it describes the design of the SystemC model of the architecture, and its runtime software library *libtmfab*.

4.1 Overview

TMFabis a many-core processor platform that incorporates a large number of processing elements, caches and other peripherals connected over the *R3* NoC. The R3 NoC consists of 7-port wormhole routers that allow the creation of 3D meshes across multiple silicon dies. The routers implement a dimension-ordered ZXY routing algorithm and offer best-effort service [15]. Components are placed within tiles containing an R3 router, and these tiles are interconnected in 2D to form one layer of the mesh. Multiple dies stacked one above the other form 3D stacked Systems-on-chip. Figure 4.1 illustrates such a die stack and the interconnection of tiles using the 3D NoC.

The TMFab architecture uses simple RISC processor cores due to their small size [14], allowing them to be integrated in large numbers at a small area cost. The processors, as previously mentioned, are also implemented within tiles. There are two types of processor tiles within the TMFab architecture:

1. *Supervising processor* (SU): The main processing tile that controls the spawning of tasks on the array, and performs input/output operations;
2. *Processing element* (PE): The processor cores, which execute parallel tasks.

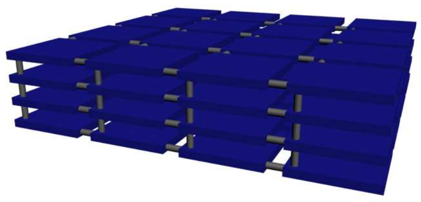


Figure 4.1: An abstract model of the 3D stacked System-on-chip.

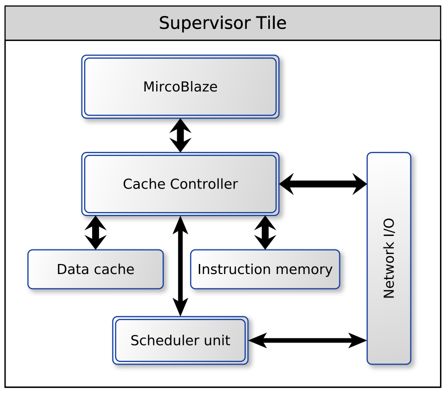


Figure 4.2: Supervisor Unit.

The architectural details of these two tiles are illustrated in Figure 4.2 and Figure 4.3, respectively. Both processor types are connected to the memory hierarchy through the 3D NoC, and access external memory through the shared cache layer, composed of multiple L2$ cache banks, each located in a dedicated tile. The total data memory address space is divided into memory pages, and distributed uniformly over the shared L2$ cache layer. Hence, each unique data address maps to only a single cache bank thus avoiding the need for a coherence scheme. The L2$ tiles are connected through an arbiter to an XRAM. Transfers with this memory are done at page granularity.

Processing Element

The primary difference between the SU tile and the PE tile lies in their internal architecture. The PE tiles incorporate transactional data caches, which enable lock-free accesses to shared data. Concurrent accesses to shared data are ordered and managed

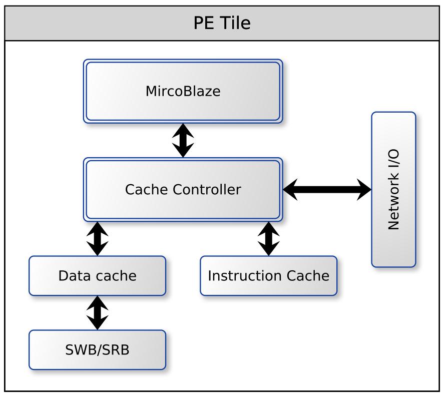


Figure 4.3: Processing Element.

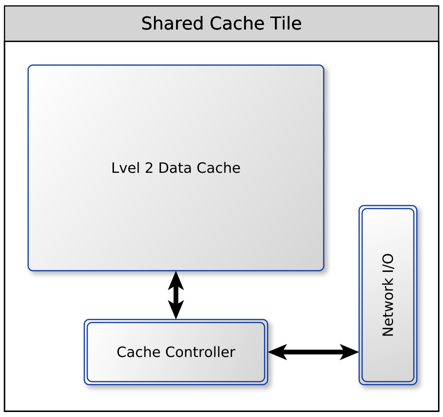


Figure 4.4: Shared cache.

automatically by TMFab's transactional memory system. Thus the PE tiles contain TM-specific hardware such as a *speculative read buffer* (srb) and *speculative write buffer* (swb). The TMFab system assumes that sequential and parallel code are not executed simultaneously. During the phase of sequential execution, all PE tiles are inactive and, similarly, during the parallel phase, the SU is stalled. The SU tile does not use the TM system and, hence, contains conventional data cache. However, since the SU tile must communicate with peripherals, IO devices and other custom components, it includes a memory mapped register set. Furthermore, since the SU is responsible for scheduling and mapping the tasks on the PEs, it includes a dedicated scheduler unit. The scheduler unit is responsible for managing the tasks in the processor array. Whenever a new task is scheduled, this unit sends out a packet to the target PE signaling that it should start a new task. It also sends a pointer to the task that must be executed, and a pointer to the data for that task. The task mapper, integrated in the scheduling unit, can map the task either on the first free processor in the array or at a target indicated in software. The SU can read the status of the different PEs in the array via memory mapped registers.

4.2 Hardware transactional memory

Hardware Transactional Memory (HTM) is used as a concurrency control scheme to enable lock-free accesses to shared data. In this scheme, processing elements execute critical sections of tasks - transactions - speculatively. The scheme assumes that no mutual exclusion is required for any accesses to shared data. However, to ensure correctness, all data used and modified by the transaction are isolated in dedicated buffers, namely the srb and swb. At the end of the transaction, the data set used/modified during execution are validated against others in the system to determine if data dependencies exist and, thus, if mutual exclusion was indeed required. If data dependencies exist, one of the contending transactions rolls back its modifications and restarts its

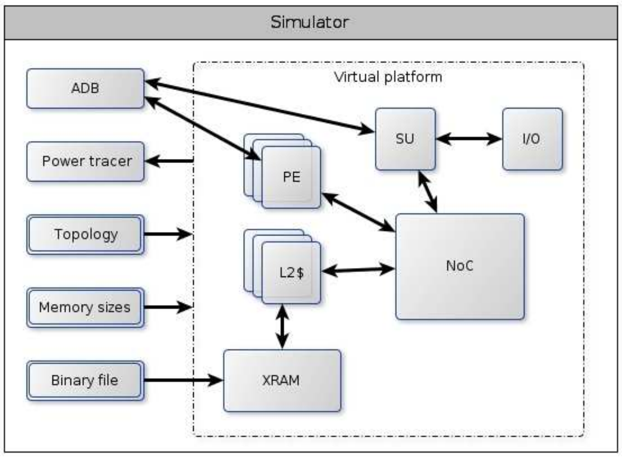


Figure 4.5: Virtual platform overview.

work, while the other transaction commits its modifications to memory.

TMFab uses the transactional memory scheme proposed by Michos et al. in [17]. In this particular implementation, cache lines read from the L2$ are appended with an additional field indicating the previous transaction that modified the line. During validation, this field is rechecked in the L2$ to determine if the cache line was modified during the time the transaction was executing. If the value is found to have changed, the validating transaction is aborted and restarted. If the value is found to be unchanged, the modifications by the validating transaction are committed to memory alongside the transaction's identifier.

4.3 Virtual platform

The TMFab architecture was implemented in SystemC [18] as a parametrizable many-core model. This allows the integration of external C/C++ based tooling with the SystemC model for debug, profiling or other purposes. An overview of the virtual platform is shown in Figure 4.5.

The virtual platform contains parameterizable simulation models for TMFab components. The PEs are implemented as single cycle instruction-set simulators that conform to the Microblaze instruction set architecture. Other components are implemented as cycle-accurate behavioural models. The platform is configured using an input specifications file that specifies the configuration and floor plan of the many-core array. Configuration parameters include:

1. L1$/L2$/swb/srb size, associativity, line size, replacement policy;
2. NoC router FIFO depth, flit width;
3. System floor plan, clock frequency.

The system floor plan is specified using a three dimensional integer array with entries corresponding to system tiles, and values corresponding to tile types. This array is parsed during elaboration to instantiate different tiles, and set up the routing tables for network interfaces.

4.3.1 Power Modeling

All simulation models within the virtual platform include a parameterizable energy model. This energy model is generated through the use of state-of-the-art tooling that reports energy per operation for various components. These values are stored in a lookup table, and scaled according to the selected clock frequency. For NoC routers, energy per flit was generated using the ORION estimator [13]. Since ORION reports lumped power values for the router, the per port estimates had to be derived from the reported values. The following formula is used to determine the active and leakage energy consumption per clock cycle per port:

$$\begin{aligned} E_{active} &= \frac{E_{load=1.0} - E_{load=0.0}}{\#ports} \\ E_{leakage} &= \frac{E_{load=0.0}}{\#ports} \end{aligned}$$

where $E_{load=x}$ is the total energy generated by ORION for a load of x and $\#ports$ is the number of input and output ports of the router. For caches, the CACTI estimator [22] was used to obtain energy per access. Since no standard estimators or energy models exist for the simple RISC cores like the Microblaze, the next procedure is used to derive the energy values per instruction. First an energy trend per instruction was generated using the Wattch [3] extension for the SimpleScalar tool [4]. Wattch uses predictive technology models to estimate power in the 90nm node. However, since SimpleScalar models the PISA instruction set, its estimates for arithmetic and logic instructions were observed to be high. Given the detailed nature of these estimates, the choice was made to scale the energy values according to a small subset of estimates available for the Leon3 processor [19]. The resulting energy model represents that of a simple embedded RISC processor core.

At runtime, the cycle-accurate activity of components, tracked through the use of activity counters, is converted into power dissipation estimates using the energy model in the form of a look up table. These can be logged as traces corresponding to the execution. The simulation setup and the obtained energy numbers are presented in Appendix A.

4.3.2 Instruction level debugging: ADB

To gain insight into the execution flow of applications executed on the virtual platform, the *assembly debugger* (ADB) was developed. ADB is designed to debug the many-core processor and allows an application developer to set breakpoint and watches at the PE granularity. To gain more insight into what happens due to the transactional memory

I/O functions
tm_printf(const char *format, ...)
tm_fopen(const char* filename, const char* mode)
tm_fclose(TM_FILE* file_stream)
tm_fprintf(TM_FILE* file_stream, const char* format, ...)
tm_fwrite(const void* ptr, size_t size, size_t n, TM_FILE* file_stream)
tm_fgets(char* dest, int size, TM_FILE* file_stream)
tm_fread(void * ptr, size_t size, size_t count, TM_FILE* file_stream)
Thread support
execute(void* taskname, void* args)
execute_on(unsigned int pe, void* task, void* args)
start_threads(tm_task_t* task_pool[], unsigned int num_task)
phase_barier()
task_barrier_create(unsigned int num_pe)
task_barrier(tmbarrier_t barrierId)
txn_start(id)
txn_end(id)
shared_read(var)
shared_write(var, value)
Memory allocation
malloc(unsigned int size)
malloc_for(unsigned int size, unsigned int pe)

Table 4.1: Function overview of libtmfab.

scheme, ADB enables the examination of values in the local L1$s, and the likelihood of success and failure of transactions. Furthermore, it supports instruction tracing and function call tracing.

4.4 libtmfab: The run-time library

A special purpose runtime software library was developed to support the various functionalities of the TMFab architecture, and to implement interfacing between the virtual platform and host machine. An overview of the function implemented in libtmfab is listed in Table 4.1.

The `execute` and `execute_on` function start a single task. The difference between the two is that the `execute` function is assigning the task on the first free PE, where as the `execute_on` function allows the application developer to specify which PE has to execute the task. The `phase_barrier` function is used to join all tasks. The SU is the only processor allowed to execute the function, causing the SU to wait until all tasks are complete. The difference between the `malloc` and `malloc_for` functions is that `malloc_for` requires the application developer to specify a PE. This is used by the memory allocator to select a memory section that maps to the closest cache bank to the specified PE.

Inside a task, the `txn_start` and `txn_end` functions are used to start and end a transaction. In a transaction all shared memory operations should be marked as such using the `shared_read` and `shared_write` functions.

4.4.1 I/O support

The platform implements a memory mapped register interface for communication between the running application and the platform. The interface is used in various functions implemented in the run-time library. The task and memory management functions, in particular, implement the principles proposed in this work, in Chapter 3. The file stream management function enables the creation and use of file streams that access files located on the host machine.

4.4.2 Parallel software model and thread support

TMFab's software model allows the SU to create and spawn a thread via a function call, in a manner similar to the Posix thread model. To provide the task mapper with better insight of the tasks that must be mapped onto PEs, the software model allows the spawning of task pools. This approach allows the mapping of tasks according to their execution characteristics.

At startup all processors executes the same instructions until they reach the main function, which is implemented in the run-time library. The main function provides all processors their own stack space, located in the nearest cache bank following the distance aware allocation scheme. Following stack space allocation, PEs wait until tasks are mapped onto them, while the SU begins with the execution of the main function of the application.

4.5 Conclusion

This section examined the TMFab transactional many-core architecture and its implementation in SystemC as a parameterizable model. The inclusion of power models for components enables the tracing of power through execution of applications, and the included assembly level debugger provides insight into the actual execution on PEs. The virtual platform is supported by a runtime software library that includes a number of functions that can be called from application software to control architectural components. This library also contains the memory and task management functions proposed in this work.

The next chapter introduces the experiments which are performed to evaluate the concept presented in the previous chapter. Also, the results of these experiments are presented in the next chapter.

Results 5

This chapter presents the results of this work as well as the simulation setup and a discussion on the measures for evaluation. The first section in this chapter discusses briefly what the values are that are measured to evaluate the quality of the work. This is followed by the system configuration and, thereafter, the results are presented.

5.1 Quality measures

In this chapter the principles presented in Chapter 3 are evaluated. These principles should reduce the access latency of the shared cache layer, and reduce the communication energy consumption. Therefore, most results are shown as execution latencies. In these numbers, the initialization latency is left out to keep the focus on how the PEs behave under certain conditions.

Next to the execution time, the number of request served by the different cache banks is evaluated. This shows which cache banks handles the memory request and, thus, what the results are of the different memory allocation schemes. The consumed energy is the final measure used to evaluate the memory allocation schemes.

5.2 Virtual platform configuration

Figure 5.1 depicts the default topology that is used for simulation, unless explicitly stated otherwise. The default virtual platform configuration is listed in Table 5.1.

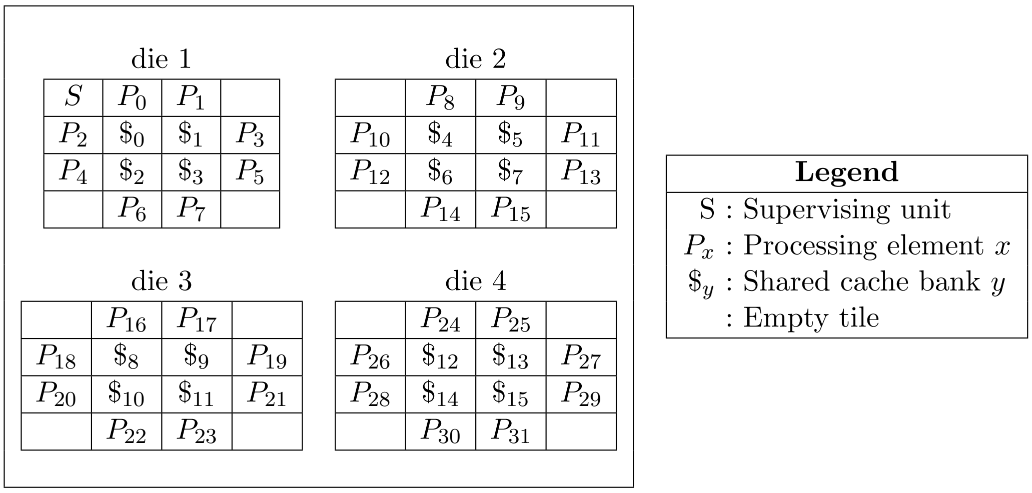
In some cases, it might be interesting to evaluate the behavior of benchmarks on the same many-core processors with a different number of shared cache banks. The number of shared cache banks limit the memory throughput for the shared cache layer to the PEs. Figure 5.2 depicts two additional topologies including 8 or 32 cache banks. In the different topologies, the L2$ associativity is selected such that the overall shared cache capacity remains the same. Thus, in case of 8 cache banks, the associativity is set to 16; in the default setup, the associativity is 8 as specified in Table 5.1. When there are 32 cache banks, associativity is set to 4.

5.3 State of the art memory allocation

In this section, state-of-the-art memory allocation is evaluated. First, the distance awareness is discussed, followed by the avoidance of cache overflow.

System configurations	
# PEs	32
# shared cache banks	16
# routers	64
Cache configurations	
L1$ size	65536 byte
L1$ associativity	4
L1$ line size	64 byte
L2$ size	131072 byte
L2$ associativity	8
L2$ line size	64 byte
# entries in srb	1024
entry size srb	10 bit
# entries swb	256
entry size swb	91 bit
NoC configuration	
# router ports	7
router fifo depth	12
virtual channels	no
NoC interface fifo depth	18

Table 5.1: Virtual platform configuration.



die 1

S	P_0	P_1	
P_2	$\$_0$	$\$_1$	P_3
P_4	$\$_2$	$\$_3$	P_5
	P_6	P_7	

die 2

	P_8	P_9	
P_{10}	$\$_4$	$\$_5$	P_{11}
P_{12}	$\$_6$	$\$_7$	P_{13}
	P_{14}	P_{15}	

die 3

	P_{16}	P_{17}	
P_{18}	$\$_8$	$\$_9$	P_{19}
P_{20}	$\$_{10}$	$\$_{11}$	P_{21}
	P_{22}	P_{23}	

die 4

	P_{24}	P_{25}	
P_{26}	$\$_{12}$	$\$_{13}$	P_{27}
P_{28}	$\$_{14}$	$\$_{15}$	P_{29}
	P_{30}	P_{31}	

Legend	
S	Supervising unit
P_x	Processing element x
$\$_y$	Shared cache bank y
	Empty tile

Figure 5.1: Default topology.

5.3.1 Distance awareness

In this section, the distance aware memory allocation is verified and evaluated. For this reason, two different synthetic benchmarks are introduced: *BlockAllocation* and *WriteArray*.

die 1

S	P_0	P_1	
P_2	$\$_0$	P_3	
	P_4	$\$_1$	P_5
	P_6	P_7	

die 2

	P_8	P_9	
P_{10}	$\$_3$	P_{11}	
	P_{12}	$\$_4$	P_{13}
	P_{14}	P_{15}	

die 3

	P_{16}	P_{17}	
P_{18}	$\$_5$	P_{19}	
	P_{20}	$\$_6$	P_{21}
	P_{22}	P_{23}	

die 4

	P_{24}	P_{25}	
P_{26}	$\$_7$	P_{27}	
	P_{28}	$\$_8$	P_{29}
	P_{30}	P_{31}	

(a)

die 1

S	P_0	P_1	P_2	P_3
	$\$_0$	$\$_1$	$\$_2$	$\$_3$
	$\$_4$	$\$_5$	$\$_6$	$\$_7$
	P_4	P_5	P_6	P_7

die 2

	P_8	P_9	P_{10}	P_{11}
	$\$_8$	$\$_9$	$\$_{10}$	$\$_{11}$
	$\$_{12}$	$\$_{13}$	$\$_{14}$	$\$_{15}$
	P_{12}	P_{13}	P_{14}	P_{15}

die 3

	P_{16}	P_{17}	P_{18}	P_{19}
	$\$_{16}$	$\$_{17}$	$\$_{18}$	$\$_{19}$
	$\$_{20}$	$\$_{21}$	$\$_{22}$	$\$_{23}$
	P_{20}	P_{21}	P_{22}	P_{23}

die 4

	P_{24}	P_{25}	P_{26}	P_{27}
	$\$_{24}$	$\$_{25}$	$\$_{26}$	$\$_{27}$
	$\$_{28}$	$\$_{29}$	$\$_{30}$	$\$_{31}$
	P_{28}	P_{29}	P_{30}	P_{31}

(b)

Legend	
S	: Supervising unit
P_x	: Processing element x
$\$_y$	: Shared cache bank y
	: Empty tile

Figure 5.2: Topologies with different L2 cache counts.

5.3.1.1 BlockAllocation

BlockAllocation is a synthetic benchmark that evaluates whether the distance aware memory allocation is working correctly. It spawns a number of threads, and every thread allocates a number of memory sections. Equation 3.1.1, the same equation used by the architecture to select a cache bank for a memory request, is used to verify whether the allocated section maps to the correct cache bank.

ptr	1	2	3	4	5
PE$_1$	2	3	4	5	0
PE$_2$	2	7	2	2	2
PE$_3$	7	0	1	3	3
PE$_4$	3	5	5	6	6
PE$_5$	4	6	6	7	0
PE$_6$	6	1	2	3	3
PE$_7$	2	2	4	4	6
PE$_8$	2	3	5	7	1
PE$_9$	7	7	0	1	1
PE$_{10}$	2	2	2	3	3
PE$_{11}$	4	5	0	2	3
PE$_{12}$	3	4	5	6	1
PE$_{13}$	2	2	3	3	4
PE$_{14}$	3	6	7	1	3
PE$_{15}$	3	4	5	5	0
PE$_{16}$	7	0	0	1	2

Table 5.2: Round robin data placement.

ptr	1	2	3	4	5
PE$_1$	0	0	0	0	0
PE$_2$	1	1	1	1	1
PE$_3$	0	0	0	0	0
PE$_4$	1	1	1	1	1
PE$_5$	2	2	2	2	2
PE$_6$	3	3	3	3	3
PE$_7$	2	2	2	2	2
PE$_8$	3	3	3	3	3
PE$_9$	4	4	4	4	4
PE$_{10}$	5	5	5	5	5
PE$_{11}$	4	4	4	4	4
PE$_{12}$	5	5	5	5	5
PE$_{13}$	6	6	6	6	6
PE$_{14}$	7	7	7	7	7
PE$_{15}$	6	6	6	6	6
PE$_{16}$	7	7	7	7	7

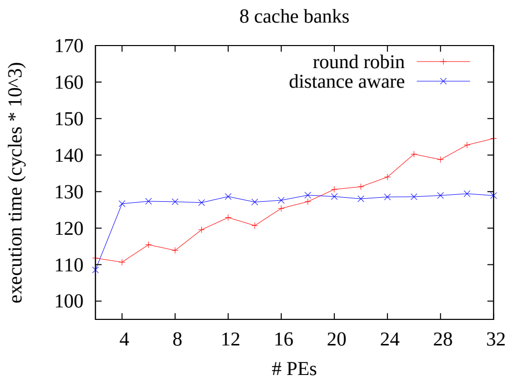
Table 5.3: Distance aware data placement.

If this benchmark is executed with 8 threads, all allocating 5 memory section, then Table 5.2 shows the outcome when allocating memory according to a round robin cache bank selection and Table 5.3 shows the outcome using distance aware memory allocation.

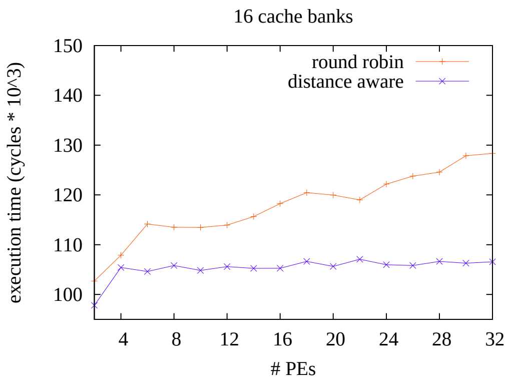
5.3.1.2 WriteArray

The second synthetic benchmark, WriteArray, is designed to show the impact of distance aware memory allocation. Every thread in the synthetic benchmark writes values into an array, where the size of the arrays are independent of the number of threads. Hence, incrementing the number of threads executing the synthetic benchmark increments the total amount of work.

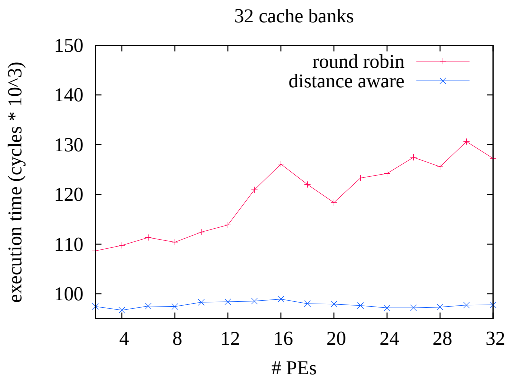
All arrays are private to the thread that writes values into the array. Therefore, changing the number of threads does not have a direct impact on the overall execution time of the synthetic benchmark. However, if data is spread over the shared cache layer using round robin in cache bank selection, data request and responses to and from the shared cache banks can potentially interfere with each other and, thus, a change in the number of threads might have an indirect impact on the overall execution time.



(a)



(b)



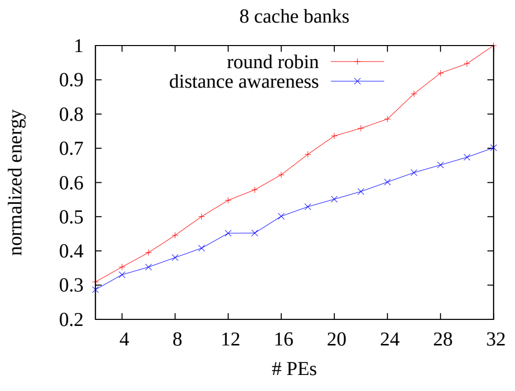
(c)

Figure 5.3: Execution time for WriteArray with distance awareness.

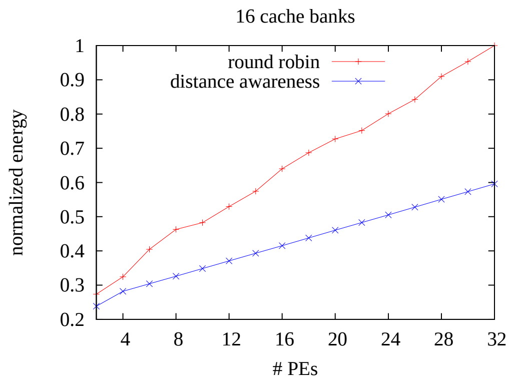
The benchmark is designed such that all arrays fit into the shared cache layer. This avoids any page misses in the shared cache layer. Also the kernel of the workload is executed twice, where the first run serves as prefetch of the arrays, and the second run as actual benchmark kernel.

Figure 5.3 presents the WriteArray execution time using a varying number of PEs. The distance aware allocation scheme is compared with round robin. The round robin bank selection shows a more or less linear increase, as expected. The irregularities in the plot can be explained by the fact that, during initialization all PEs try to allocate their private arrays at the same time. Hence, which PE allocates first varies with the number of PEs and, thus the hop-count in the communication paths between the shared cache and the PEs varies as well. The flat lines for the distance aware allocation shows that this principle has indeed a significant impact. The comparison between the round robin and distance aware allocation shows that a larger system, with potentially more active PEs, has a bigger significance.

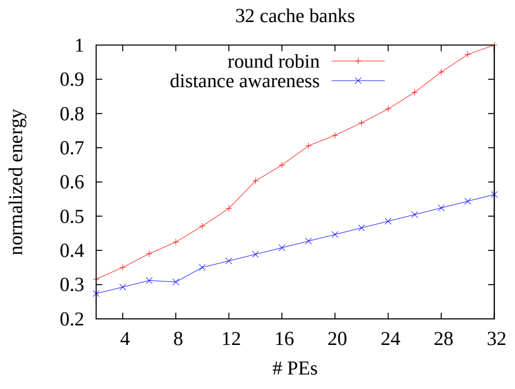
Figure 5.11 presents the total NoC energy for the execution of the WriteArray kernel on a varying number of PEs. These graphs show that there is a significant difference in the energy consumed by a round robin scheme compared to the distance aware scheme.



(a)



(b)



(c)

Figure 5.4: NoC energy for WriteArray with distance awareness.

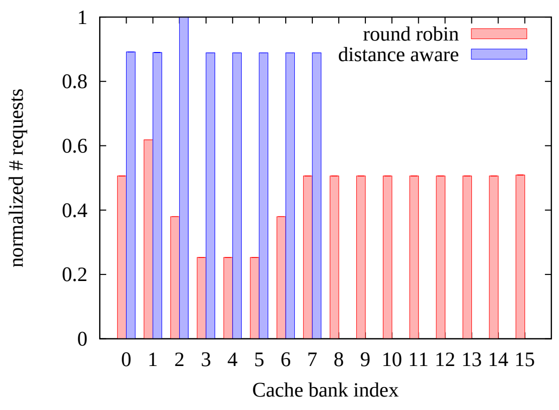


Figure 5.5: Requests for WriteArray with distance awareness.

The different PE to cache bank ratios show, that when the PEs execute an application that heavily utilizes the shared cache banks, it might result in hot caches. When the number shared cache banks is equal to the number of PEs, there is no change when the number of active PEs is increased. However, when this ratio changes, a significant difference presents itself when PEs start to share cache banks.

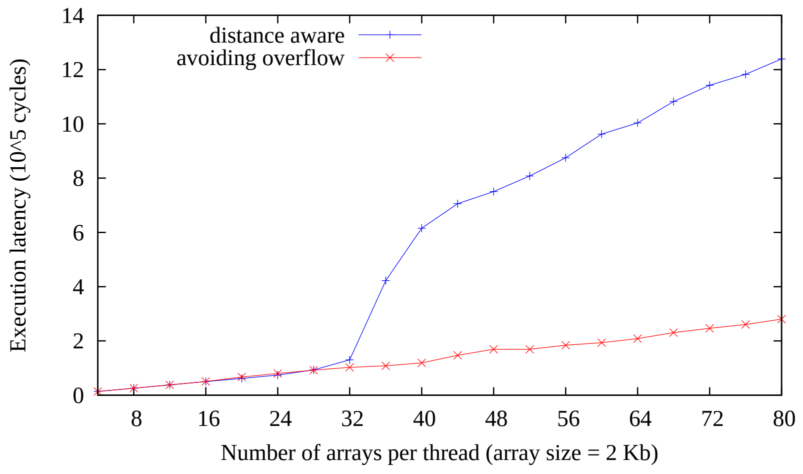


Figure 5.6: Page dispersion due to cache overflow.

Figure 5.5 shows the request distribution to the different caches. This graph reveals that the distance aware scheme allocates memory such that it maps to a limited number of cache banks.

5.3.2 Page dispersion due to cache overflow

To analyze the impact of page dispersion caused by cache overflow, the synthetic benchmark CapacityTest is used. This benchmark is an extension of the WriteArray benchmark, which implements one very important difference. The array size used by the threads is variable and, possibly, larger as the capacity of a shared cache bank. Figure 5.6 compares the execution time for distance awareness against the memory allocation scheme implementing page dispersion due to cache overflow. As all PEs are forced to use memory that only maps to the cache bank closest to the PE, eventually the cache bank runs out of capacity and is forced to use off-chip accesses to the XRAM. While the distance aware scheme runs out of capacity the cache overflow avoiding scheme is allowed to disperse pages and avoid the off-chip accesses to the XRAM.

Figure 5.7 shows the distribution in memory request, for all the cache banks. This figure reveals that the used memory is indeed spread over more cache banks if page dispersion is applied.

5.4 Cache Balancer

This section evaluates the quality of the Cache Balancer. The first part of this section focuses on the memory allocation scheme. Thereafter, the task mapping is evaluated and, finally, the memory allocation and task mapping schemes are compared to state-of-the-art.

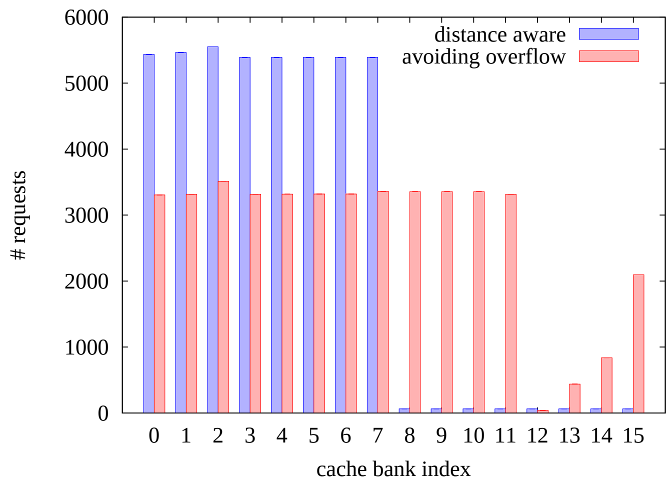


Figure 5.7: Data distribution due to cache overflow avoidance.

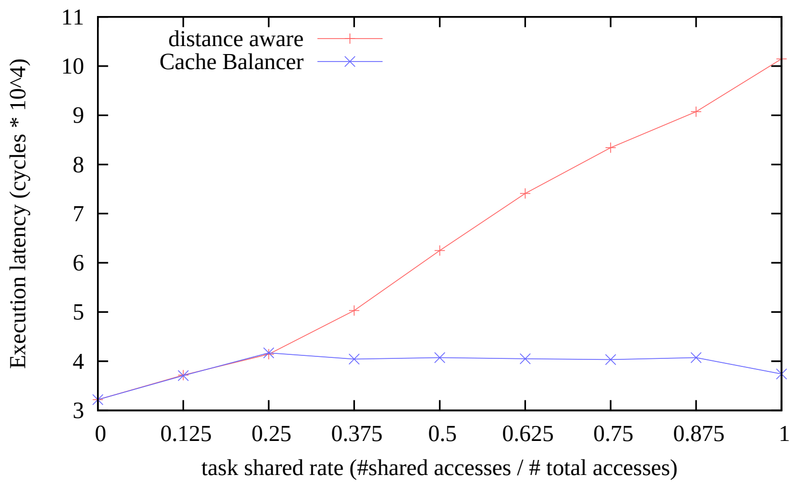


Figure 5.8: Hot cache avoidance.

5.4.1 Utilization aware memory allocation

The *AccessControl* synthetic benchmark is used to show the effect of hot caches. This benchmark uses 7 tasks to control the access rate of a single cache bank. The tasks have an equal shared and private memory section available. In the kernel of the benchmark the tasks write, according to a given shared to private ratio, to either the private or the shared data. The shared data is shared by all tasks and mapped to a single cache bank. If the shared to private ratio is high, the cache bank becomes hot. If the ratio is decreased, then the cache becomes less hot.

A victim task is placed next to the potentially hot cache. The task allocates a memory section and writes values into the allocated section. If the cache bank close to the victim task is hot, the task takes longer to execute, since the task has to wait more often before a memory request is served.

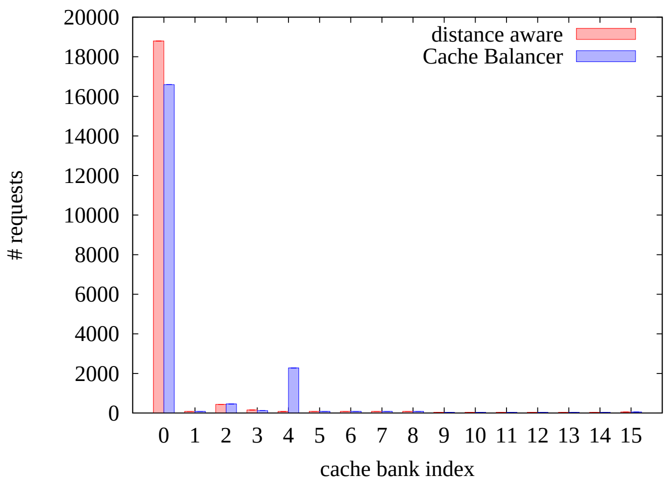


Figure 5.9: Cache access due to hot cache avoidance.

Figure 5.8 depicts a plot of the execution latency of the victim task. In this plot the hot cache avoidance scheme is compared with distance aware. The reduction in execution time is mainly due to a reduction in memory access latency, which is a reduction of 63.4%. Figure 5.9 compares the number of requests per cache bank for the case where the hot cache access is 1.0. Even though the change is little, the memory access latency is significantly reduced when the allocated memory maps to a cache that is cold.

Figure 5.10 depicts the different execution latencies for the WriteArray benchmark, which is the same benchmark used to evaluate distance awareness. This benchmark consists of two different phases: an initialization phase and a kernel phase. The memory allocation takes place in the initialization phase, causing a certain cache access pattern. Thereafter, the kernel is executed, causing a different cache access pattern compared to the initialization phase. The Cache Balancer uses the access rate as prediction for the future. Figure 5.10b and Figure 5.10c show a little worse performance than the distance aware memory allocation scheme because the two different phases have different cache access patterns. Figure 5.10a reveals that, when the number of cache banks is small compared to the number of PE, the cost of the miss prediction is overcome by the benefit of the increased memory throughput caused by Cache Balancer. The energy consumption change in the NoC caused by Cache Balancer is insignificant, as shown in Figure 5.11.

The execution of the WriteArray benchmark with 16 tasks combined with the Cache Balancer results in a request distribution, as depicted in Figure 5.12. This histogram shows how the Cache Balancer distributes the data over the memory, and thereby reduces contention in the shared cache banks. The result is that the energy is spread over more cache banks compared to the distance aware scheme, as shown in Figure 5.13.

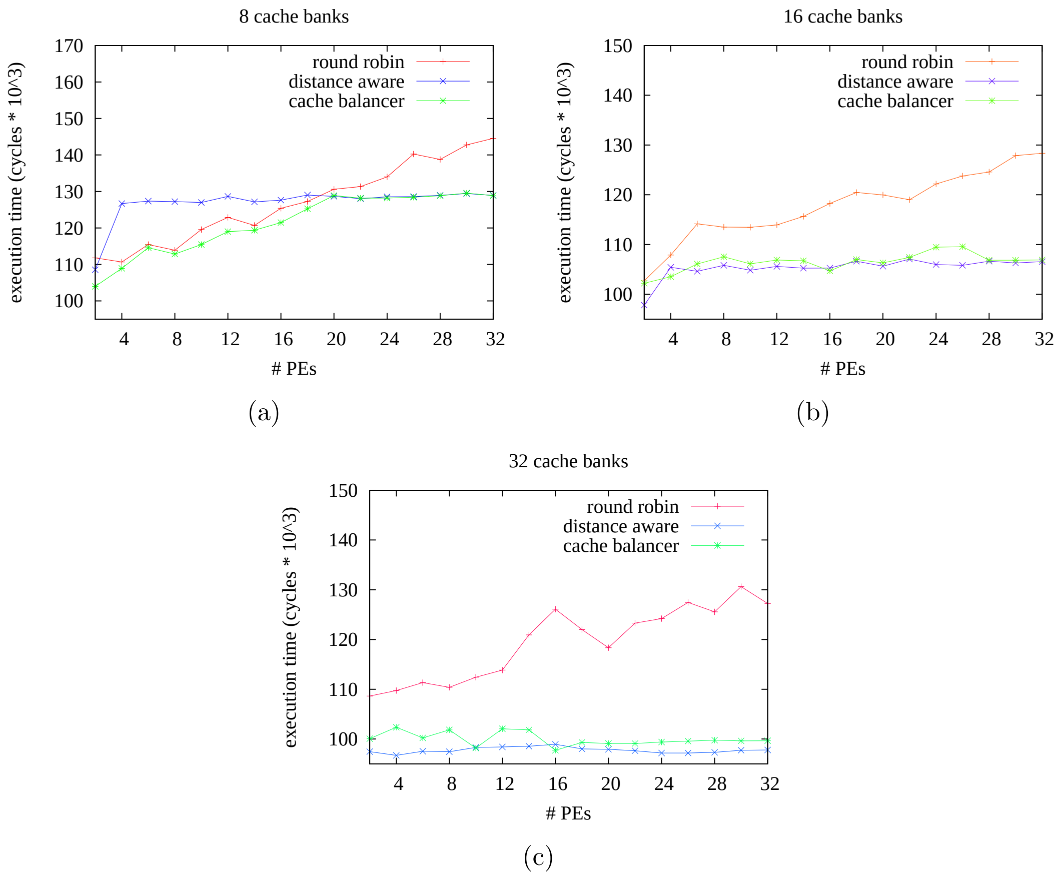
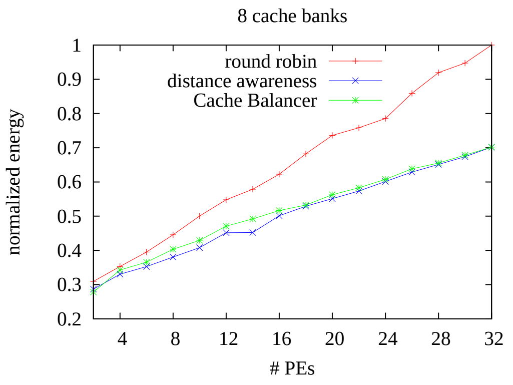


Figure 5.10: Execution time for WriteArray with Cache Balancer.

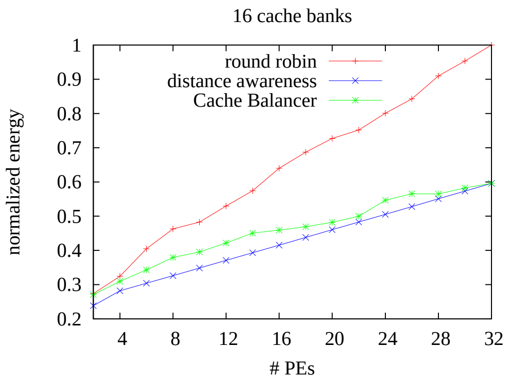
5.4.2 Pain driven task mapping

Figure 5.14 presents execution time for the WriteArray benchmark when the tasks are mapped by Cache Balancer. The stepwise incrementing execution time that especially stands out in Figure 5.14a is a result of used task pain. Because all used data in the tasks is private, the task pain is causing the Cache Balancer to map the tasks such that they do not share, as less as possible, the cache banks. Thus if only 8 task are mapped in a many-core processor with 8 cache banks, the task are mapped such that they do not share any cache bank. Up until 16 tasks, Cache Balancer maps the tasks such that only two processors share a cache bank. This effect is still visible in Figure 5.14b. However, when the number of cache banks is equal to the number of PEs, tasks are always mapped to a PE that does not share a cache bank.

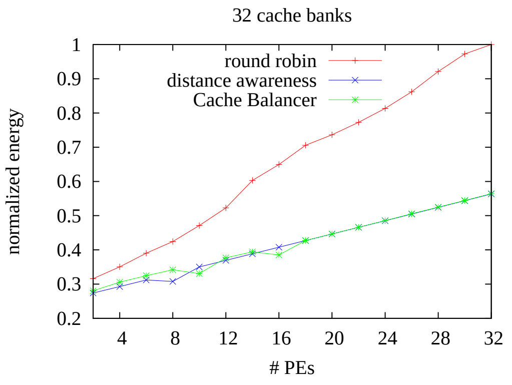
Figure 5.15 depicts the NoC energy consumption for the WriteArray benchmark where tasks are mapped by Cache Balancer. The slight decrease in energy consumption with a small number of active PEs, as shown in Figure 5.15a, is because the task are mapped closer to the cache bank compared to round robin task mapping. Also the consumed leakage energy is less, since the execution time is decreased.



(a)



(b)



(c)

Figure 5.11: Energy for WriteArray with Cache Balancer.

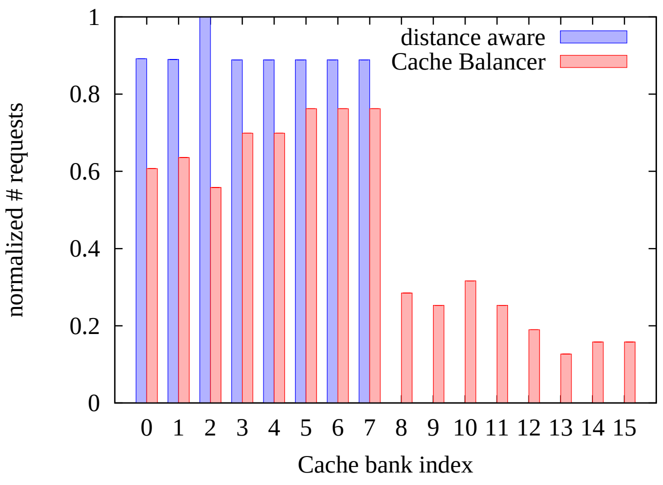


Figure 5.12: Requests for WriteArray with Cache Balancer.

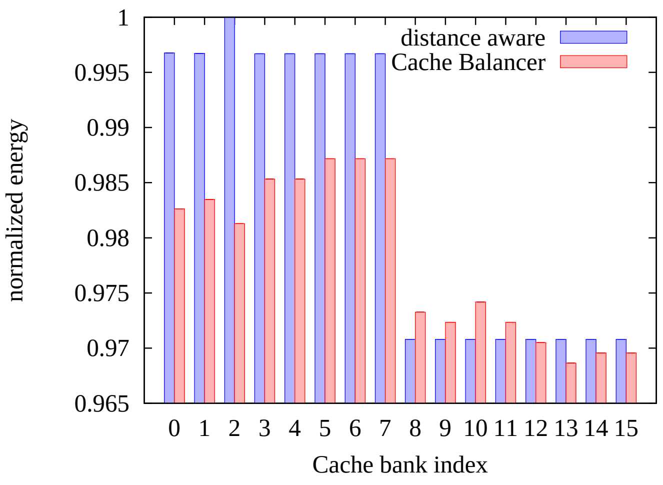


Figure 5.13: Cache energy density for WriteArray with Cache Balancer.

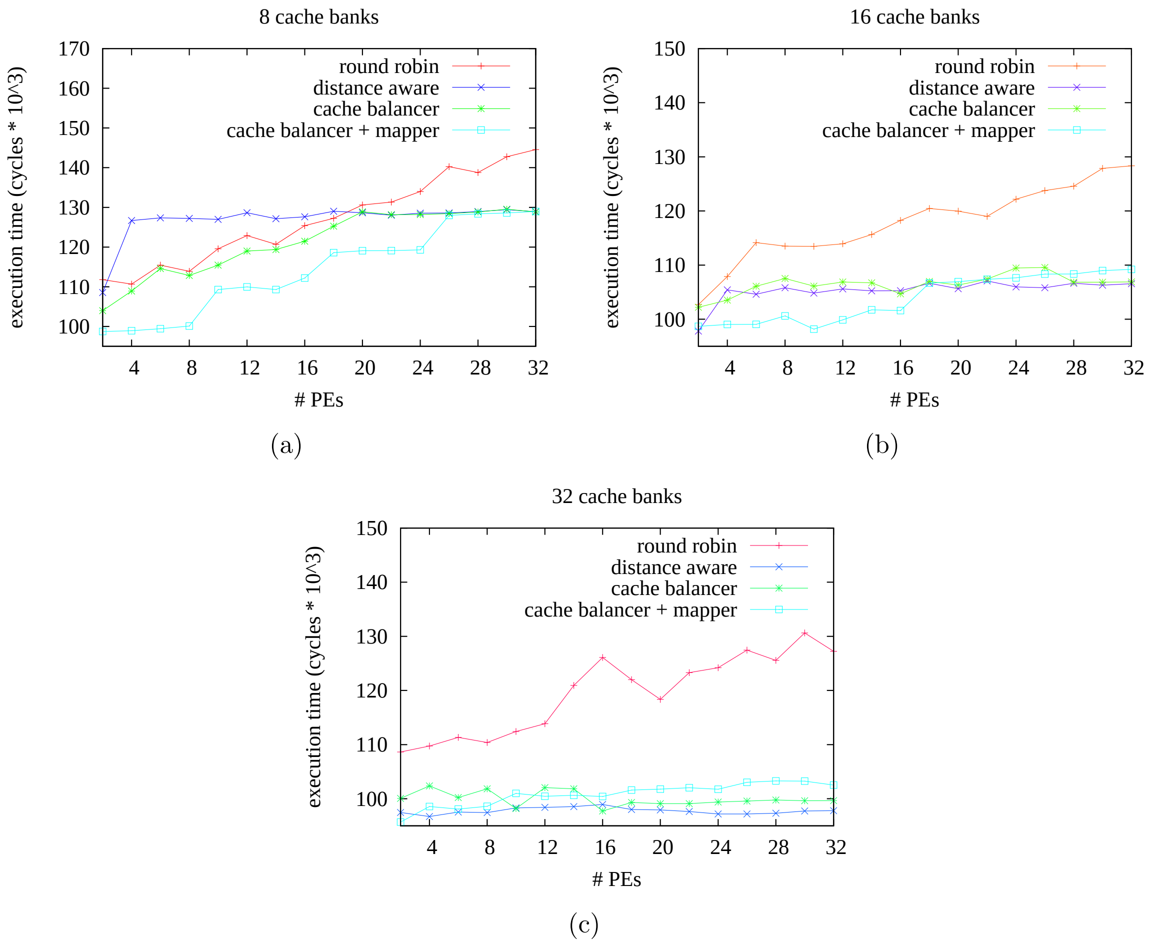
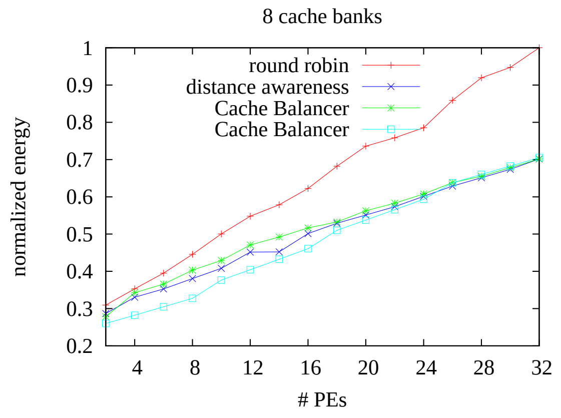
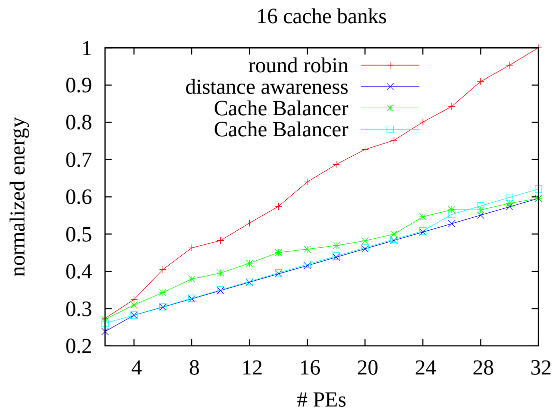


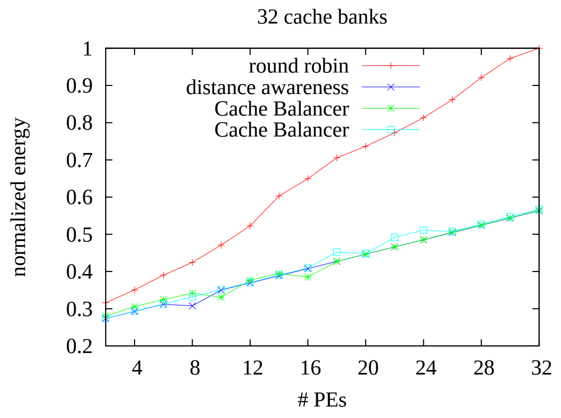
Figure 5.14: Execution time for WriteArray with Pain driven task mapping.



(a)



(b)

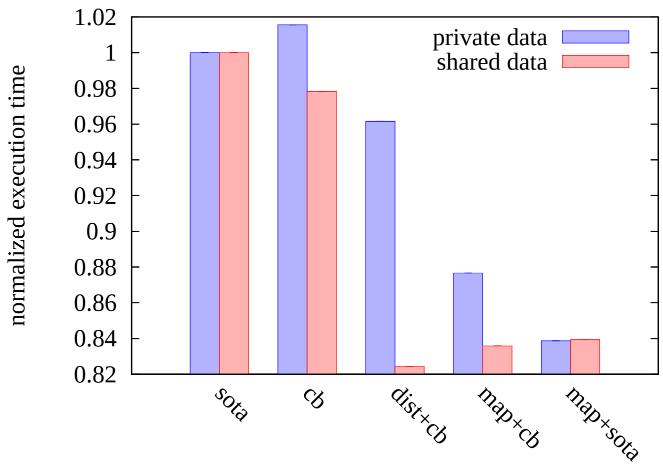


(c)

Figure 5.15: Energy consumption for WriteArray with Pain driven task mapping.

To demonstrate the different effect from Cache Balancer, a synthetic benchmark is used, similar as to the AccessControl benchmark, used to evaluate the memory allocator. Again, multiple tasks either share a cache bank for shared data, or use private data in separate cache banks. The benchmark is executed with different components of Cache Balancer switched off, which are then switch on one at the time. Because the topology with 8 cache banks showed the largest difference, this topology is used to evaluate the different components. To demonstrate the effect of a large data set without increasing the simulation time significantly the shared cache bank size is reduced to 64Kb. The different task maps are shown in Table 5.4.

Then next chapter presents the conclusion of these results and, with that, the results of this work.



sota	:	State-of-the-art memory allocation
cb	:	Cache Balancer memory allocation
cb+dist	:	Cache Balancer with distance aware task mapping
cb+map	:	Cache Balancer with both task mapping and memory allocation
sota+map	:	state-of-the-art memory allocation with pain driven task mapping

Figure 5.16: The different effects in Cache Balancer.

num	**t1**	**t2**	**pr**
0	0	1	2
1	3	4	5
2	6	7	8
3	9	10	11
4	12	13	14

(a) Round robin

num	**t1**	**t2**	**pr**
0	3	27	7
1	2	31	6
2	0	29	8
3	4	19	5
4	1	28	9

(b) Distance aware

num	**t1**	**t2**	**pr**
0	8	27	5
1	4	20	13
2	0	19	16
3	12	21	9
4	6	28	24

(c) Pain driven task mapping with shared data

num	**t1**	**t2**	**pr**
0	1	0	4
1	8	12	16
2	20	24	28
3	5	9	17
4	21	29	6

(d) Pain driven task mapping with private data

Table 5.4: The different task maps.

6 Conclusions and future work

6.1 Conclusions

This work has explored state of the art memory allocation and task mapping, to investigate whether this can reduce memory access latency in many-core processors. In the introduction, two hypotheses where defined, speculating that the average memory access latency is reduced when the communication costs are minimized, while keeping a fair balance in the used capacity and utilization of the different caches.

Several steps have been taken to investigate the impact of the different effects in both memory allocation and task mapping. As first step, the concepts of state-of-the-art memory allocation is implemented for a many-core processor. The scheme integrates both distance awareness and used cache capacity information in a memory allocation scheme. The memory allocator is extended, allowing it to consider cache utilization as well. To obtain information about the cache utilization, this work introduces access rate, which represents how heavily a cache is utilized relative to the other caches. A measurement system for the access rate was integrated into the virtual platform, allowing the memory allocator to use this at run-time.

The second step involves a task mapping scheme. An algorithm, which considers cache utilization while tasks are mapped to the PEs, is used as starting point. Because this algorithm was designed for an architecture where PEs could share only a single cache, the algorithm was extended in such a way that all caches used by a PE are considered. Thereafter, distance awareness was integrated into the algorithm.

The last step was the development of a virtual platform. This platform simulates a parameterized many-core processor. The parameterization allows the selection of different topologies and different memory sizes. The system simulates two different data cache levels, a shared L2$ and a private L1$. In the shared cache layer data replication is prevented to avoid coherence overhead. The private caches use a hardware transactional memory scheme, which eliminates the need for a coherency scheme.

The different memory allocation and task mapping algorithms are compared to each other to obtain insight into the impact of the different effects. The results showed that distance awareness in memory allocation reduces memory access latency compared to round robin memory allocation. A second important observation is that the latency of an increased communication distance is almost insignificant compared to the latency of a L2$ miss. Hence allocating memory that maps to a cache further away compared to the closest one to avoid L2$ misses can reduce the memory access latency significantly. Similarly, if communication latency is increased to avoid the allocation of memory that maps to an overutilized cache reduces memory access latency. Avoiding the overutilized caches with memory allocation can reduce the memory access latency by 63.4%, and this confirms the first hypothesis. Finally, the last part of Chapter 5 showed that the same effects should be considered during task mapping. After an in depth analysis of the mapping algorithm, it was found that this can further reduce execution time up to 14.5%, and this confirms the second hypothesis.

6.2 Future work

Two different topics are of interest for future work regarding this thesis. First, this work does not take thermal effects in consideration and, thus, dynamic voltage and frequency scaling is not used. If this is considered, then:

- caches close to PE running at a low frequency are less utilized, allowing PEs running at a high frequency to use these caches, which could prevent the other caches from becoming over utilized;
- the more uniform utilization of the resources spreads the power dissipation out over a larger area, which could lead to lower temperatures.

The second topic concerns task or application scheduling. In this work, only a single application is considered. However, it is likely that a many-core processor executes multiple applications concurrently. Then, it might be the case that it is more efficient not to schedule all tasks at the same time, but to reserve some resources for other applications. A second reason not to schedule all tasks at the same time might be that the increased performance is very little compared to the additional used energy.

6.3 Publications

- Jurrien de Klerk, Sumeet S. Kumar, Rene van Leuken; **CacheBalancer: Access Rate and Pain Based Resource Management for Chip Multiprocessors**, submitted to Computer Systems & Architectures workshop, International Symposium on Computing & Networking 2014.
- Demonstrator: DATE 2013 University Booth

A Energy look up tables

A.1 Network on chip modules

General NoC setup	
flit width	36
virtual channels	0
output buffer	disabled

(a)

Router setup	
flit width	36
virtual channels	0
output buffer	disabled

(b)

Network interface setup	
Number of input ports	1
Number of output ports	1
Input FIFO depth	18

(c)

Table A.1: Orion NoC simulation setup.

Router energy (pJ)	
E_{active}	19.4277
$E_{leakage}$	2.1017

(a)

Network interface en. (pJ)	
E_{active}	13.3933
$E_{leakage}$	2.1297

(b)

Table A.2: NoC energy look-up table.

A.2 Memory hierarchy modules

Setup	
Total size	131072 bytes
Associativity	8
Bus width	88
Tag size	42

(a)

Energy (pJ)	
E_{active}	200.267
$E_{leakage}$	263.511

(b)

Table A.3: Level 2 data cache energy look-up table.

ICache	
Total cache size	4096 byte
Associativety	2
Tag size	21
Bus width	64

L1DCache	
Total cache size	65536 byte
Associativety	4
Tag size	74
Bus width	138

SRB/hazard table	
Total cache size	32768 byte
Associativety	direct mapped
Tag size	10
Bus width	20

SWB	
Total cache size	16384 byte
Associativety	direct mapped
Tag size	66
Bus width	26

Table A.4: Processor local memory module configuration.

Energy (pJ)	
$ICache_{leakage}$	9.7664
$ICache_{access}$	21.1452
$L1DCache_{leakage}$	143.3645
$L1DCache_{access}$	143.015
$SRB_{leakage}$	1.6149
SRB_{access}	2.9629
$SWB_{leakage}$	34.0547
SWB_{access}	41.6812

Table A.5: Processor local memory module energy look-up table.

A.3 Microblaze processor

logic & arithmatic	
and	103.0716
or	103.4004
xor	103.4004
nor	102.9254
add	100.7102
addi	100.0037
sub	100.8526
subi	99.9312
mult	103.1342
multi	103.6681
srl	101.3231
sra	101.7831
cmp	100.0353
cmpu	100.3451

load/store	
lbi	218.4289
lb	219.1571
lhi	231.0722
lh	231.7597
lwi	231.0722
lw	231.0722
sbi	276.9693
sb	256.7231
shi	235.1698
sh	235.4715
swi	235.4959
sw	235.5801

branch	
bri	237.7202
bril	295.5496
br	199.9647
brl	257.7941
beq	213.3156
bne	213.2385
ble	207.6671
bgt	207.5900
blt	179.7809
bge	225.5001

float	
fadd	221.0395
frsub	221.3111
fmul	224.6329
fdiv	242.8718
fcmp_eq	221.9587
fcmp_lt	221.9587
fcmp_le	221.9169
flt	221.0813
fint	221.0813
fsqrt	260.6720

Table A.6: Microblaze instruction energy in pJ.

Bibliography

[1] T. Agarwal, A. Sharma, A. Laxmikant, and L.V. Kale. Topology-aware task mapping for reducing communication contention on large parallel machines. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10 pp.–, April 2006.

[2] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. *SIGPLAN Not.*, 35(11):117–128, November 2000.

[3] 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.

[4] Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, June 1997.

[5] A. Chahar. Compile time analyses for hardware transactional memory architectures. Master's thesis, TU Delft, 2012.

[6] Sangyeun Cho and Lei Jin. Managing distributed, shared l2 caches through os-level page allocation. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 455–468, Washington, DC, USA, 2006. IEEE Computer Society.

[7] Chen-Ling Chou and R. Marculescu. Contention-aware application mapping for network-on-chip communication architectures. In *Computer Design, 2008. ICCD 2008. IEEE International Conference on*, pages 164–169, Oct 2008.

[8] Chen Ding and Yutao Zhong. Predicting whole-program locality through reuse distance analysis. *SIGPLAN Not.*, 38(5):245–257, May 2003.

[9] Dongrui Fan, Hao Zhang, Da Wang, Xiaochun Ye, Fenglong Song, Guojie Li, and Ninghui Sun. Godson-t: An efficient many-core processor exploring thread-level parallelism. *Micro, IEEE*, 32(2):38–47, March 2012.

[10] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Reactive nuca: Near-optimal block placement and replication in distributed caches. *SIGARCH Comput. Archit. News*, 37(3):184–195, June 2009.

[11] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[12] Lei Jin and Sangyeun Cho. Sos: A software-oriented distributed shared cache management approach for chip multiprocessors. In *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*, pages 361–371, Sept 2009.

[13] Andrew B. Kahng, Bin Li, Li-Shiuan Peh, and Kambiz Samadi. Orion 2.0: A fast and accurate noc power and area model for early-stage design space exploration. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 423–428, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[14] Tamar Kranenburg and Rene van Leuken. Mb-lite: A robust, light-weight soft-core implementation of the microblaze architecture. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 997–1000, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

[15] S.S. Kumar and R. Van Leuken. A 3d network-on-chip for stacked-die transactional chip multiprocessors using through silicon vias. In *Design Technology of Integrated Systems in Nanoscale Era (DTIS), 2011 6th International Conference on*, pages 1–6, April 2011.

[16] Zoltan Majo and Thomas R. Gross. Memory management in numa multicore systems: Trapped between cache contention and interconnect overhead. *SIGPLAN Not.*, 46(11):11–20, June 2011.

[17] A. Michos. A novel concurrent validation scheme for hardware transactional memory. Master's thesis, TU Delft, 2012.

[18] P.R. Panda. Systemc - a modeling platform supporting multiple design abstractions. In *System Synthesis, 2001. Proceedings. The 14th International Symposium on*, pages 75–80, 2001.

[19] S. Penolazzi, L. Bolognino, and A Hemani. Energy and performance model of a sparc leon3 processor. In *Digital System Design, Architectures, Methods and Tools, 2009. DSD '09. 12th Euromicro Conference on*, pages 651–656, Aug 2009.

[20] A. Ros, M. Cintra, M.E. Acacio, and J.M. Garcia. Distance-aware round-robin mapping for large nuca caches. In *High Performance Computing (HiPC), 2009 International Conference on*, pages 79–88, Dec 2009.

[21] Lingjia Tang, Jason Mars, and Mary Lou Soffa. Contentiousness vs. sensitivity: Improving contention aware runtime systems on multicore architectures. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 12–21, New York, NY, USA, 2011. ACM.

[22] S. J E Wilton and N.P. Jouppi. Cacti: an enhanced cache access and cycle time model. *Solid-State Circuits, IEEE Journal of*, 31(5):677–688, May 1996.

[23] Y. Zhong, S.G. Dropsho, and Chen Ding. Miss rate prediction across all program inputs. In *Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on*, pages 79–90, Sept 2003.

[24] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. *SIGPLAN Not.*, 45(3):129–142, March 2010.

Acronyms

ADB *assembly debugger*. 39

L1$ *level 1 data cache*. 5, 6, 12–15, 19, 26, 30, 34, 35, 37, 39, 42, 45, 49

L2$ *level 2 data cache*. 5, 6, 12, 14–20, 34–37, 42, 44

lhoard *light hoard*. 36

LRU *least recently used*. 34

NoC *network on chip*. 5, 11–13, 31, 33, 38, 42, 45

PE *processing element*. 12–20, 33, 35, 36, 42

srb *speculative read buffer*. 14, 18, 19, 34, 42

SU *supervising processor*. 12–15, 18, 20, 35, 36

swb *speculative write buffer*. 14, 17, 19, 35, 42

TMFab *Transactional Memory Fabric*. 11, 12, 49

XRAM *external random accessible memory*. 9, 14, 33, 35, 46, 47