Task Centric Memory Management for an On-Chip Multiprocessor

Task Centric Memory Management for an On-Chip Multiprocessor

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof.dr.ir. J.T. Fokkema,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen

op vrijdag 30 januari 2009 om 10:00 uur

door

Anca Mariana MOLNOŞ

inginer
Universitatea Politehnica Bucureşti, Roemenië
geboren te Făgăraş, Roemenië

Dit proefschrift is goedgekeurd door de promotor:

Prof.dr. K.G.W. Goossens

Copromotor:

Dr. S.D. Coţofană

Samenstelling promotiecommissie:

Rector Magnificus, voorzitter	Technische Universiteit Delft
Prof. dr. K.G.W. Goossens	Technische Universiteit Delft, promotor
Dr. S.D. Coţofană	Technische Universiteit Delft, copromotor
Prof. dr. ir. H.J. Sips	Technische Universiteit Delft
Prof. dr. h.c. mult. M. Glesner	Technische Universität Darmstadt
Prof. dr. ing. N. Tăpuş	Universitatea Politehnica Bucureşti
Prof. dr. ir. J.A.G. Jess	Technische Universiteit Eindhoven
Dr. ir. M.J.M. Heijligers	NXP Semiconductors
Prof. dr. ir. C.I.M. Beenakker	Technische Universiteit Delft, *reservelid*

ISBN: 978-90-9023974-3

Keywords: Cache memory, Multimedia applications, Compositionality

To my family and friends,
for all their support over the years.

Task centric cache management for an on-chip multiprocessor

Anca M. MOLNOŞ

Abstract

In this dissertation we propose a cache memory management method for embedded chip multiprocessors executing multimedia applications with soft-real time constraints. We consider a CAKE multiprocessor platform with 4 TriMedia cores and an on-chip memory hierarchy including a shared level two (L2) cache; we assume a multimedia workload. The interference of concurrent accesses to the shared L2 is typically unpredictable. Hence the system has to be compositional to offer real-time guarantees, i.e. the performance of each individual sequential task must be preserved when different tasks execute in parallel or when tasks are added. To ensure compositional cache access we propose to exclusively allocate L2 parts to each task. We compare set and associativity based L2 partitioning for applications with independent tasks and find that both induce compositionality to a large extent: the number of inter-task L2 interference misses is within 1% of the application's number of misses. However, the former increases, and the latter decreases the system performance. Thus, we propose set-based partitioning as the foundation for the task centric cache management. For applications with dependent tasks, we introduce a mixed partitioning that allocates cache set-based to each task and each shared data/code region, and associativity-based inside a shared data/code region, for each task accessing it. Subsequently, we propose two methods for partitioning ratio optimization, one to minimize the number of misses and one to maximize the throughput. Experiments indicate that mixed partitioning ensures less than 1% inter-task L2 interference, (i.e. high compositionality) and improves performance. Finally, for multiple execution scenarios when the application tasks may start or stop we propose a dynamic cache repartitioning method that: (1) at design-time finds the cache place of each task in every scenario, such that critical tasks are undisturbed, and the repartitioning overhead is minimized, and (2) at run-time switches among partitions and further decreases the repartitioning penalty. Experiments show that this method induces high compositionality, safeguards critical tasks, and increases performance. The results obtained indicate that task centric cache management is a promising approach for embedded multiprocessor systems executing static or dynamic multimedia applications.

Acknowledgments

First, I would like to pay my respects to two professors that shaped my professional life, and that are not anymore among us. I am indebted to Prof.Dr. Stamatis Vassiliadis, who was my promotor in the first years of my Ph.D work. I regret I did not have the opportunity to spend more time around Stamatis, to learn more from his vast computer science experience, and from his impressive science and history knowledge. Next, I would like to thank Prof.Dr. Irina Athanasiu, who introduced me to the world of compilers. Furthermore, without her support I would not have had the fortunate chance to start a Ph.D track at T.U. Delft.

Secondly, I would like to thank my professors from Universitatea Politehnica Bucureşti, Romania, who laid the foundations of my computer knowledge and academic interests. I especially want to mention Nicolae Tapuş and Cristian Giumale, who influenced my professional but also personal choices, and to which I am in debt.

Next, I wish to thank Prof.Dr. Kees Goossens, my current promotor, for his sharp and quick feedback in all matters. Though I met Kees only toward the end of my Ph.D. track, I have learned something from each of our discussions, which positively influenced me and my thesis. Thank you Kees!

I am especially grateful to Dr. Sorin Coţofană, my copromotor, for his continuous guidance in my research. During the years, he patiently spent a lot of time with me, always bringing energy, new insights, and ideas. To not be forgotten, besides shaping my academic thinking and technical writing, he introduced me to new flavors of music, plastic arts, and culture in general, and he spiced everything up with some Romanian language ”delights”. Many thanks Sorin!

From Philips and NXP, the companies I work with in the last years, I am especially in debt to my supervisor at Philips, Dr. Marc Heijligers, to Dr. Jos van Eijndhoven my contact person with the CAKE team, and to the leaders of the groups I was part of, Dr. Frans Theeuwen, and Dr. Ad ten Berg. Marc offered

me not only technical and personal guidance, but also very useful insights in the Dutch society. Further, he always trusted me, even when I doubted myself. Jos introduced me to the CAKE multiprocessor world, supported me and always answered my technical questions in great detail. Frans helped me in arranging the Ph.D. related funds at Philips and Ad gave me the time to finalize my thesis while working at NXP. Thank you all!

I have the great luck to have as friends many exceptional people, and they influenced directly or indirectly my thesis. I am in debt to my dear friend Erwin Woutersen, with whom I shared great fun, who continuously encouraged me all these years, and showed me some sides of positive thinking I did not know before. Thanks Erwin! Moreover, I wish to thank the following: my very close friend Ana Vârbănescu for our endless personal and scientific data sharing that enriched my life and increased the quality of this thesis; my colleague and good friend Bart Vermeulen - not only that his technical and linguistic knowledge are reflected in this thesis, but he managed to cheer me up even in my ”yach”-est late-work evenings; my dear friend George Moldovan for his almost daily, warm, presence in my life (despite the 2000km that part us), and for turning all my holidays in Romania into some crazily fun experiences; my dear friend Eugen Tibelea together with which, at the age of 14, we started to share our scientific curiosity, while wondering how can black-holes exist (we still did not find that out); Hillien Pinxterhuis for her contagious good mood, and for being a great walking, talking, and shopping friend. The limited space prevents me for expressing the deserved gratitude nominatively. However, I would like to mention some friends that brought me fun and inspiration: Oana, Paul, Sorin, Corina, Niels, Karina, Laura, Cristina, Dana, Irina, Alexandru, Cătălin, my basketball friends, and my T.U. Delft, Philips, NXP officemates and colleagues. The others, please accept my non-nominative thanks.

Last but not least, I wish to thank my family. Mom and dad thanks for always encouraging me to chase my dreams (even when those dreams were not that realistic), for your love and continuous warm support. I am also very grateful to my brother Radu. His special ways of expressing his love and trust spared me for ever feeling lonely, but also kept me sharp and skilled in avoiding collisions with flying objects.

The journey toward a Ph.D. degree has its ups and down. Mine might have been lost in its downs without you all. Thank you! Bedankt! Mulţumesc!

Anca Molnoş Delft, The Netherlands, 2009

Contents

Chapter 1

Introduction

Traditionally, computing is intended to automate mathematical calculations, for solving large engineering problems or predicting nature's behavior. However, due to a dramatic decrease in the cost of computing power, data processing is nowadays embedded in virtually every electronic device. Even though such a device is not generally considered to be a computer, it encapsulates a processing unit, called an embedded system. Unlike a general-purpose processing unit, such as the one found inside a personal computer, an embedded system performs one or a few pre-defined tasks, usually with very specific requirements. While no unique definition exists for "embedded systems", it is widely accepted that such a system represents a combination of computer hardware, software, and perhaps additional parts (e.g. mechanical, electrical, etc. subsystems), designed to perform a dedicated function. Examples of embedded systems range from the "traditional" ones, like industrial robots, household devices, etc., to the novel ones like automobile driving assistants, personal digital assistants, mobile phones, video recorders, electronic toys, etc.

Because of the environment in which they are used, embedded systems have functional and performance requirements, as well as specific technical and economical constraints. First, embedded systems often implement functionality that has to be performed in *real-time*, i.e. the output has to be delivered within fixed and (sometimes) critical time deadlines. Some typical examples are video, audio applications and automotive systems. As missing a deadline can cause severe quality degradations and may have critical consequences, an application's timing has to be guaranteed by construction. To enforce this guarantee, the behavior of the system has to be *predictable* in

each context the application might execute. Furthermore, a system may be composed by a number of parts which execute concurrently, and which might be developed by independent teams. If the interference among these parts is random, their integration is difficult, and the predictability of the system is endangered. Therefore, the performance of each individual part must be preserved even if the parts are executed concurrently, in arbitrary combinations, or if new parts are added to the system. A system satisfying this property is addressed as being *compositional*. Second, many embedded systems are portable, e.g., mobile phones, digital assistants, pacemakers, etc. Hence, heat dissipation, weight, and size matter, imposing severe *low resource constraints* (e.g., for memory, energy, etc.). Third, it is desirable that an embedded system is *dependable*, meaning that it continuously delivers an acceptable level of service, regardless of any internal or external disturbance. Moreover, no possible system state should be able to provoke catastrophic consequences on the users or/and the environment; also the system should have the ability to undergo modifications and repairs. Last but not least, the number of players on the embedded system markets has increased in the last decade, resulting in growing pressure for shorter *time to market* and *low cost*.

Concomitantly with their rapid proliferation, the embedded devices feature more and more functionality and options. For example, multimedia features, like video decoders or sound players are integrated nowadays in most portable devices. Moreover, a user might regard it as a normal technology evolution to enjoy high quality video images on a device like a mobile phone or a personal digital assistant. However, the designers of such systems are facing serious design challenges to cope with the constraints and requirements imposed by these explosions of features.

Besides the general embedded systems constraints, multimedia applications are typically demanding a *huge amount of computation power*. Consider, for instance, the H.264 video standard of the International Telecommunication Union [46]. To process in real time a high definition video stream, an H.264 video decoder requires at least 50 Giga Operations Per Second (GOPS) [4]. This is quite a design challenge knowing that a modern, high performance, processor like IBM PowerPC 970 delivers only 12.5 GOPS, four times less than the demand of H.264 decoding.

As for every processor, the performance of a multimedia processor is dictated by two factors: the ability to execute fast computation (processor speed) and the immediate availability of the operands (memory latency). On the computation side, considering that the operands are always available, a processor's

absolute performance is given by the product of two factors: (1) the number of operations that it can execute in a basic unit of time (an interval known as ”clock cycle”), and (2) the length of this time unit. Consequently, when a platform cannot provide the required performance, there are two options for speedup: increase the number of operations executed in a cycle, or shorten the cycle time (increase the processor's clock frequency). However, the energy dissipated by a processor varies quadratically with its frequency, thus a frequency increase results in an power consumption increase [45]. In the aforementioned example, the IBM PowerPC 970 operating at a frequency of 1.2 GHz consumes at least 40 watts [40], while delivering only a quarter of the required computation performance. In contrast, allowing the user to watch one-hour of video on a single mobile phone battery, a processor should utilize just few watts, or even less. Thus, increasing the frequency of a IBM PowerPC processor to cope with the computation requirements is an unacceptable solution in terms of power consumption.

As a result of these specific low power constraints, speeding up embedded workloads is achieved mostly by concomitantly executing more instructions every cycle. In turn, this is possible only if the application exhibits parallelism. Fortunately, multimedia applications (or shortly media applications) intrinsically have a large degree of parallelism [94]. Therefore, such an application is split into groups of instructions, further called ”tasks”, and these tasks are executed in parallel on multiple processing units. These processing units might have various complexities (from simple functional units to fully fledged processors). Due to the large amount of transistors that can be nowadays integrated on-chip [45] multiple such units can be embedded on a single chip.

In the above processor speed discussion, we abstract from the data operands availability. However, in practice, it is rare that the all operands are available in each processor cycle, as technological restrictions limit the capacity of the storage units accessible in one cycle. In fact many media applications process data amounts so large that they can entirely fit only in the off-chip main-memory. In any case, the data access time plays an important role in an application execution time. In general, the latency of accessing a memory block is inversely dependent on its storage capacity [45] and its distance from the processor. Nevertheless, the processor speed increases with 60% per year, whereas the memory speed increases only with 10% per year [38]. This leads to a growing speed gap of 50% per annum between processor and memory speeds. To mitigate this gap, a common practice is to buffer a part of the data on the chip where the processing units reside, thus reducing the data ac-

cess latency. These buffer levels, together with the main-memory are known as ”memory hierarchy”. Such a hierarchy can ameliorate the memory speed deficit if applications exhibit the so-called locality of reference: data and instructions are accessed multiple times within a short interval [88]. Therefore, a good memory allocation strategy is to keep the often accessed items in a small but fast buffer close to the processor (commonly denoted as ”high level” or ”level one” memory). The items that are needed less often are stored on lower hierarchy levels that are larger and slower. In this dissertation we focus on the memory hierarchy organization of a multimedia embedded system and we target parallel platforms with multiple processor cores.

As mentioned before, for embedded media processing, the system performance has to be *predictable*. For the memory hierarchy, predictability requires the latency of each data access to be known. To determine the latency of an access one has to know on which hierarchy level the data reside, and how much time it takes to access that level. Hence, the location of each data item should be known in every clock cycle. As a consequence, the designers would like to explicitly regulate the traffic through the memory hierarchy (so called scratch-pad memory organization). Because we consider applications consisting of concurrent tasks, enforcing a strict data traffic control, each instruction inside a task has to be analyzed for each possible combination of tasks. If some instructions change (standards updates and the addition of new features, or tasks occur often in the multimedia domain), the instruction level as well as the task level analysis have to be reiterated. Moreover an application may have many utilization scenarios, which have to be all detailed and separately investigated. Such a large design effort clashes with the short time to market demand specific to the embedded applications area. In order to build profitable devices, application parts have to be reused, and updates have to be performed during the lifetime of the device (on the fly). In other words, the design itself, as well as the design procedure have to be *flexible*. This is difficult in an environment where every small change implies the redesign of the entire system. Ideally, in a flexible system the memory traffic should be implicitly controlled by the hardware (cache organization), and completely transparent to the applications which are using it, such that the application can be updated independently of the memory system. Although the conventional cache organizations are flexible, they are known to be unpredictable [13, 95], which bans their straightforward utilization in embedded systems.

Summing up the technical and economical considerations mentioned thus far we can conclude that, besides the ”classical” constraints like (high performance, predictability, low resource consumption, low cost, fast time to market,

and dependability), *flexibility* (in terms of potential parts reuse on the fly updates) is equally important. In fact. flexibility versus predictability is a critical trade-off in embedded processing. On one side, fully predictable systems are rigid and take a lot of effort to be designed, and on the other side, fully flexible systems are highly unpredictable.

In this context, this thesis focuses on the memory hierarchy organization of an embedded system consisting of a set of processors residing on a single chip. As a suitable workload, we study state-of-the-art multimedia applications consisting of multiple tasks. For such applications, our main concern is to bridge the processor-memory gap while realizing a good predictability-flexibility trade-off.

1.1 Related work and problem statement

The increasing speed gap between the processing units and the off-chip storage resource has been forecasted decades ago [74]. Therefore, a significant amount of research has been carried out in the domain of memory hierarchy organizations. With respect to the on/off-chip memory traffic control there are two major options: (1) it can be explicitly controlled by the programmer or the compiler (a so-called scratch-pad memory organization) or (2) it can be implicitly controlled at run-time (cache). In the following we present advantages and shortcomings of both these options:

(1) *Scratch-pad*. As already mentioned, due to the predictability requirements of the embedded systems, a desired execution scenario is to explicitly control the data traffic through the memory hierarchy. In the case of scratch-pad memories, the overall data residing on-chip is under tight in control every clock cycle [6], [8], [27], [44], [49], [54], [83], [112]. When data have to be processed, they are loaded on the chip by explicit instructions inserted in the application code. This implies that at each and every small change of the software, the entire application and its utilization scenarios should be reanalyzed and the memory traffic redesigned. This analysis is especially difficult in the case of a multi-processor executing a multi-tasking application because task scheduling and mapping have to be taken into account when determining which data reside on-chip at a given moment in time [53]. Besides the time and effort needed for the analysis, this inflexibility results in poor portability across multiple hardware platforms. Moreover, common programming techniques like dynamic memory allocation and data address manipulation may not be supported because all data sizes have to be known at design-time. We conclude

that, by its very nature, the scratch-pad organization does not fulfill the flexibility requirement. Therefore, it is not suitable as starting point in our quest for a memory organization that can provide opportunities for predictability-flexibility trade-offs.

(2) *Cache*. Implicitly controlled memories (caches) do provide flexibility to the memory hierarchy. In a cache organization, a hardware controller performs run-time data lookups in the on-chip buffers and it is responsible for the data traffic. The overall data traffic through the memory hierarchy is transparent to the software, therefore when the applications are updated, no redesigning or limitations are present. Under these conditions, a larger variety of applications can be easily used, with little porting effort. However, caches hinder the software predictability. For instance, when a task needs its data on-chip, the implicit control mechanism may decide to swap other task's data off-chip to create enough free space. If the second task requires its swapped data, the previous replacement decision may cause a future, expensive, off-chip access. This cache flushing may depend on the exact timing details of each task, on tasks memory access pattern, and on tasks input data. Such information is not easily available at design-time, making the prediction of the inter-task cache interference practically impossible. This kind of unpredictable conflict misses constitutes a major problem for real-time applications for which the completions of tasks before their deadlines is of crucial importance.

Different approaches that attempt to make caches usable in real-time environments already exist in the literature. These approaches fall into two categories: (1) attempts to estimate the software tasks' cache behavior, and (2) attempts to partition the caches among tasks such that they become predictable. In the following we comment on both these directions.

Most of the existing work in estimating the cache behavior of multiple tasks running on an embedded platform [62, 81, 48, 107, 111] is applicable only to single processors, because they can only cope with time multiplexed tasks execution and not with truly concurrent tasks execution. All these studies assess the cache interference penalty due to task switching; for multi-processor systems (our target platforms), we consider them of very limited relevance. To the best of our knowledge, few articles focus on the in-depth quantitative investigation of cache sharing in a multiprocessor architecture [11], [17], [19], [30], and [100].

In [11], the authors compare the cache performance for the same computation load in two cases: sequential execution and parallel execution. Due to cache contention among parallel units, the off-chip data traffic is larger in the

parallel execution than in the sequential execution of the same computation. Intuitively, to reach the same amount of off-chip traffic, the parallel execution has to benefit from a larger cache than the sequential one. This paper further provides an analytical framework to calculate the size of this extra cache. This method is based on a restricted computation model. The authors assume that a computation is represented as a Directed Acyclic Graph (DAG) of tasks synchronized with barriers and scheduled in a depth-first manner. In [1] and [30] the authors propose a similar technique for analyzing the cache complexity but for the case of distributed caches. Their techniques also assume applications described as series-parallel DAGs, but this time scheduled according to a Clik work-stealing scheduler [12]. The authors of [19] extend the work of [11] to cover the case of larger cache sizes, and experimentally investigate the effects of both work stealing and parallel depth first scheduling algorithms on cache performance, for different tasks granularity. Their study suggests that parallel depth first scheduling is cache friendly, delivering better performance than work stealing, for a sorting workload. Nevertheless, real life media applications are difficult to express or implement using DAGs scheduled in such specific ways (depth-first or Clik work-stealing). Furthermore these approaches do not directly target compositionality and take into account only an ideal cache model, making their method difficult to apply to the applications we target.

The work in [17] predicts the inter-task cache contention based on the cache profile of each task. The main conclusion of this article is that cache contention can cause a significant performance penalty to the concurrently executed tasks. In turn, this suggests that, even with a method to estimate the penalty of cache sharing, the parallel cache access is still problematic as its effects go beyond predictability, negatively affecting memory performance. Therefore the research in [17] makes the point of the present thesis even stronger, as it highlights the need for methods to manage caches in a multiprocessor environment.

A method to reduce the inter-processor cache conflicts and improve cache performance is introduced in [100]. Here data are locked in the cache parts that suffer the most inter-processor cache flushes. In order to determine at runtime which are those ”hot” cache parts, the authors introduce an accounting scheme for the ownership of cache parts. Whereas this method increases the cache performance, the compositionality is not its target.

The second manner to deal with cache unpredictability, cache partitioning, was investigated by several research groups. In general, cache partitioning

has been proposed to: (1) improve performance [20], [78], [90], (2) achieve predictability [47], [58], [63], [43], and (3) obtain fair cache sharing [57]. In the following we briefly comment the currently available cache partitioning schemes, highlighting the reasons why they are unsuitable or insufficient for our targeted context. We first discuss the methods that establish the cache partitions at design-time (static partitioning methods) and then the ones that change the partitions at run-time (dynamic partitioning methods).

In [20] the authors consider the case when a system consists of two types of tasks: with and without real-time characteristics. In the proposed scheme, every real-time task has an exclusive cache part that works as a scratch-pad memory while all non-real-time tasks shared a single cache part. Furthermore, in [103] the same authors tackle the problem of optimal cache allocation between two competing processes such that the overall miss rate is minimized. Their method is based on a cache partitioning scheme with a small number of available resources, therefore only a limited number of tasks can be accommodated. The processor speed up due to this type of cache partitioning is explained in detail in [86], for different workloads. The authors introduce application metrics that allow the designer to decide if this cache partitioning has potential in improving the application performance. However, the fact that current media applications have a much larger number of tasks makes this partitioning scheme unsuitable for them.

The authors of [43] and [78] propose a compositional data (and instructions, respectively) cache organization. After the application is analyzed in detail, a specific partitioning is decided at design-time and imposed at run-time by specific cache instructions. The main drawbacks of this approach are very similar to the ones of scratch-pad memories. The analysis that can be performed at design-time is limited, and difficult for the multiprocessor case.

In [90] the authors propose to vary the associativity of a cache on a per-set basis in response to the demands of the program. Although this is principially similar with cache partitioning, the cache cannot distinguish among different tasks, thus compositionality is not supported.

In [77] the cache is partitioned among tasks at compile and link time. In [58] the authors propose to divide for the real-time tasks. For non-real-time tasks a shared cache pool is provided. The authors of [63] propose the cache partitioning to be controlled by the operating system. In [47] the authors discuss a quality of service like cache management strategy for multiple memory access streams. In [57] the authors introduce a cache partitioning strategy to improve fairness (defined as how uniform tasks are slowed down due to

cache sharing). All these solutions bring interesting ideas to the field, but they all assume that the tasks are independent. This assumption does not hold true for media applications where sharing of large data structures (like video frame buffers, for instance) is very common. Therefore, none of the available cache partitioning approaches are suitable for real-life media applications running on multiprocessor platforms.

As the challenges of slow background memory and limited off-chip bandwidth become more acute with the increased number of processors integrated on a chip, the topic of dynamic cache management receives more and more attention. The state-of-the-art dynamic cache repartitioning solutions have two major targets: power reduction and performance improvement. In the following we first discuss the dynamic cache partitioning methods oriented toward single processors, like [3], [7], [15], [51], [56], [79], [87], [92], [106], [118] and then continue with the methods tailored for multiprocessors like [28], [39], [84], [89], [97].

In [51] the authors study the benefits of a hardware-software co-adaptation scheme where the cache configuration and the optimization strategy are modified at run-time. The cache configurations and optimizations for each loop nest are determined at compile-time; further the right re-configuration instructions are inserted in the code.

In [15] Cai et al. estimate the impact of cache size selection for reducing energy consumption and enhancing reliability for time constrained systems. The conclusion of this work is that different programs have different cache sizes that give the best performance-energy-reliability trade-off, advocating dynamic cache reconfiguration as a beneficial technique.

Zhang et al. [118] propose way-concatenation, a technique called that can change four cache parameters: cache line size,cache size, associativity, and cache way prediction. The cache reconfiguration is performed at run-time, and it is based on a tuning heuristic that can automatically adjust the cache to an executing program aiming at performance improvement and energy saving. Their approach is extended in [36] to also support the management of a second level of unified cache for which the ways can be specified as a unified way (instruction and data), an instruction-only way, a data-only way, or the way can be shut down entirely.

The authors of [106] introduce a prioritized cache for real-time systems in which tasks considered critical may temporarily utilize a larger number of cache ways than other tasks. Using the prioritized cache, the worst case execution time of a task can be estimated more precisely. Moreover, the miss rate

of the tasks that have priority in the cache decreases.

In [87] the authors propose to modify the structure of a configurable cache to offer embedded compilers the opportunity to reconfigure it according to a program's dynamic phase. They use the way-concatenation scheme introduced by [118] but the reconfiguration is at the granularity of program phases.

Balasubramonian et al. [7] propose a cache configuration management algorithm that dynamically detects program phase changes and reacts to the number of hits and misses in order to improve the memory hierarchy performance, while still taking energy consumption into consideration.

Nacul et al. [79] propose a dynamic on-line scheme that combines processor voltage scaling and dynamic cache reconfiguration. Two cache configurations are different form one another by at least one of the following configurable parameters: cache size, line size or cache associativity. The algorithm operates at run-time, in two phases. The first phase (the Pareto-discovery phase) is designed to discover the performance of each task under different cache configurations. The second phase is the cache configuration selector.

In [56], the authors investigate means for splitting an instruction cache into several smaller units, each of which is a cache in itself (called a subcache). The proposed subcache architecture employs a page-based placement strategy, a dynamic page remapping policy, and a subcache prediction policy in order to improve the energy behavior of the memory system.

Ranganathan et al. propose in [92] a reconfigurable cache strategy for media applications. Different parts of the cache are used for different processor activities (the partitions could be used as hardware look-up tables, for instruction reuse, or as storage area for prefetched information). Their experimental results suggest that instruction reuse coupled with reconfigurable caches leads to computation performance improvement.

Albonesi presents in [3] a possibility to disable a subset of the ways for program regions with modest cache activity, in order to reduce the energy consumption. The program regions with modest cache activity are detected at run-time, and energy can be saved while having a small performance degradation. The energy-performance trade-off is flexible as it can be dynamically tailored.

All these methods provide some degree of performance improvement and/or power saving. However they do not address the multiprocessor domain, or the application compositionality.

In the multiprocessor domain, Hsu et al. [39] investigate three types of

cache management policies that can be applied to a shared cache of a chip multiprocessor (CMP). The utilized workload consists of multithreaded, general purpose programs from the SPEC suite. The first policy targets equal performance for each thread (labeled as ”Communist”), the second one targets the improvement of overall system performance (labeled as ”Utilitarian”), and the last is the conventional, free-for-all, shared cache model (labeled as ”Capitalist”). The paper investigates different performance criteria (miss rate, instructions per cycle, etc.), and concludes that there are large performance variations among the different policies for different workloads; based on these results, the authors, suggest that thread aware cache allocation is required for getting good performance from a CMP.

Petoumenos et al. [84] propose a statistical model to estimate the cache performance and a partitioning mechanism to improve the performance of a shared cache accessed by a CMP. The model estimates at run-time the behaviour of each thread, while the control mechanism dynamically assigns poorly used cache parts to threads that are likely to benefit the most from them. A cache partitioning method targeting a similar CMP platform is presented in [89]. In this paper Qureshi and Patt categorize applications as having low, high, and saturated cache utility, and utilize run-time utility monitoring hardware as the input for a look-ahead cache partitioning algorithm. This algorithm evaluates every possible partitioning decision and decides which one can potentially deliver a performance improvement.

Chang and Sohi [18] propose a CMP cache partitioning method in which each thread does not use only a single partition, but shares several partitions with other threads. In this manner, a fair cache balancing among threads can be achieved, while improving the system throughput.

In [28], the authors propose a non-uniform cache architecture in which the amount of cache space that can be shared among the processors is set dynamically. The purpose of this partitioning scheme is to increase the overall multiprocessor throughput, and the paper reports significant speedups when compared with the existing, conventional schemes.

In [97] the authors propose a dynamic hardware cache management for programs consisting of multiple threads. Hardware counters are employed to measure the reuse of each cache line. Using this information and information regarding the amount of cache each task has, the paper presents a method to compute the effect of removing a thread from a cache way. Based on the gain vs. loss estimation the cache partitioning is dynamically changed. This technique balances cache demand of each task and improve the overall system

throughput.

However, none of these approaches targets compositionality and critical tasks performance protection, as these solutions are positioned mostly in the general purpose computation area.

In the view of the previous discussion, this thesis addresses the problem of finding an on chip memory organization which can ”feed” data to a multiprocessor such that:

- The system is **flexible** in the sense that it supports parts reuse and update without requiring the redesign of the entire application.
- The task's **predictability** is preserved. Multimedia applications may comprise of both tasks with and without real-time constraints. The performance of the real-time tasks has to be predictable and should not be disturbed by the behavior of other tasks in the system. From the memory perspective, the traffic parameters of each real-time task should be guaranteed, and protected against the interference with other tasks that might be executed concurrently, thus the memory access has to be compositional.

For this purpose we assume a multiprocessor platform containing an on-chip cache hierarchy. On such a platform each and every processor core may have its own cache memory (called L1 cache in this dissertation). As these L1 caches cannot provide the required application bandwidth [102], a shared level two (L2) cache is also provided [80], [114]. To ensure flexibility and compositional memory access we propose a novel task centric cache management. We apply this management technique to the L2 cache, as this level is accessed in parallel by multiple processors, hence it is the most affected by inter-task conflict misses. Our cache management strategy falls in the category of cache partitioning and the detailed research questions that we answer are the following:

- **Which are the possible options for cache partitioning and which one of them is the most suitable to media applications?** Starting from a conventional cache organization, the possibilities to partition a cache have to be identified. Any partitioning method can potentially ensure compositionality, but the performance of different cache splitting methods might be different. Therefore, we have to investigate which of the potential partitioning options is more appropriate for media applications.

- **How to achieve cache compositionality in a typical media application consisting of parallel tasks that may exchange data and/or share instructions?** For achieving compositionality tasks should interfere as little as possible with each-other. Task based cache partitioning could solve this problem. However, in typical media applications, multiple tasks share data and/or instructions. For example, data sharing in media applications occurs when multiple tasks perform different processing steps on the same video frame. In this case, keeping separate copies of the frame is highly inefficient. Similarly, code sharing is common in media applications where the same processing is done by several tasks on different input data, e.g., different frames.

  The interference inside such a shared data or instructions region is intrinsic, posing an extra challenge to the cache management system.

- **How much cache should be allocated to each task such that the execution time, and/or throughput are optimized?** The compositionality problem is solved in principle by cache partitioning. Therefore, the system has an extra optimization parameter, the cache partitioning ratio. This ratio has to be determined such that a relevant criterion for the multimedia domain is optimized.

- **Is the cache management method robust to application's variations?** After predictability, robustness is another desired property of an embedded system, directly related with dependability. The lack of robustness is an important issue in media applications where the tasks completion before a deadline has to be guaranteed. In this context two types of robustness are relevant: internal (performance deviations are caused by the application tasks comprising) and external (performance variations are caused by external stimuli). Whereas predictability is improved by compositionality, the question whether the resulted system is robust requires a separate answer.

- **How to ensure cache compositionality for the case when the system is dynamic, in the sense that tasks are started and stopped at runtime?** Typically, state-of-the-art embedded media devices require the starting and suspending of tasks. Allocating a part of the cache to each task, even when it is suspended, might not be beneficial from a performance point of view, as it leads to low resource utilization. Thus, there is a need for a cache reconfiguration strategy that implies a low reconfiguration penalty. Moreover it is interesting to know how the performance of the cache varies with the application's reconfiguration frequency.

1.2 Thesis overview

This section introduces the organization of the remainder of this dissertation which consist of the following chapters:

- In Chapter 2 we present the background and the terminology used in this thesis, followed by a description of the targeted multiprocessor platform (the CAKE multiprocessor [114]), its memory hierarchy, and programming model. Furthermore, we present the utilized benchmark applications that embody memory intensive programs from MediaBench [22] as well as state-of-the-art media workloads relevant for the industry. We utilize applications consisting of (1) non-communicating and (2) communicating tasks. The applications consisting of non-communicating tasks are formed by running several MediaBench programs in parallel. For the applications consisting of communicating tasks, we use two industry-relevant media applications, a picture-in-picture mpeg2 decoder and a H.264 decoder. These applications are used for the experiments presented throughout the thesis.

- In Chapter 3 we identify two options for cache partitioning (namely set-based and associativity-based) that can potentially lead to system compositionality. We propose a new implementation method for the set-based partitioning that does not require compiler modifications, and it is not dependent of the memory addressing model. We assess the compositionality and the performance of these schemes on a practical implementation of a CAKE multiprocessor, utilizing non-communicating tasks applications. We find that both partitioning schemes can ensure compositionality within 1% bounds. Moreover, the experimental results reveal that the L2 misses per instruction delivered by set-based partitioning is 55% smaller, on average, than those generated by associativity-based partitioning, and 29%, on average, smaller than the L2 misses per instruction of the conventional shared cache. This leads to an average application speedup of 27% and 8%, when compared to associativity based partitioning and to a conventional shared cache, respectively. These results recommend task centric set-based partitioning as the best candidate for a cache management scheme.

- Chapter 4 introduces a novel, task centric cache management method tailored for typical media applications. This method consists of a static cache partitioning scheme and an optimization strategy. The partitioning

scheme is responsible for achieving a high compositionality in a media application consisting of tasks that share data and/or instructions. Subsequently, the optimization strategy determines the cache partitioning ratio that minimizes the number of misses or maximizes the throughput for a partitioned cache. Applying the proposed method to the communicating tasks applications, we observed that the amount of inter-task interference is under 1%, suggesting that the proposed cache partitioning ensures compositionality to a large extent. When the platform was equipped with a relatively small cache size, our simulations indicate that the smallest number of misses achievable by a partitioned cache is larger than the one of a shared cache of the same dimensions. Concretely, the misses per instruction of a partitioned cache are 17% larger (resulting in a 6% cycles per instruction increase) than the ones of a conventional cache. The reason for the increase in the number of misses is that the experimented applications have a large number of tasks and common regions, hence the static partitioning induces a high cache fragmentation, leading to performance degradations in the case of small caches. When the throughput maximization method is applied, the throughput is improved with 7% at the cost of 14% misses growth. The reason for this growth is the fact that the throughput optimization strategy tends to give more cache to the tasks that are on the application critical path, and that does not necessarily minimizes the application's number of misses. Thus, for these applications, for small caches, the price of compositionality is a degradation in performance. For average cache sizes the partition that minimizes the number of misses has, on average, 5% less cycles per instruction and 28% less misses per instruction than the shared cache. The partition that maximizes the throughput has more or less the same performance as the one minimizing the misses. For large cache sizes the performance of the partitioned and shared caches are very close. Summing it all up, for average size and large caches, the compositionality comes with a performance increase too.

- To evaluate the robustness of a system that is using our cache management we introduce in Chapter 5 two new metrics to assess: (1) the performance variations caused by the application tasks (internal robustness) and (2) performance variations caused by external stimuli (external robustness). According to our experiments, the performance variations due to internal interferences are below 8% and the variations due to external factors are below 10%. These numbers prove that the system is robust in the presence of cache partitioning.

- In Chapter 6 we extend the task centric cache management with a dynamic method for applications with multiple execution scenarios, and critical tasks. In particular, we propose a method to solve the compositional caching problem for the case when the tasks of the application may start and/or stop at run-time. In this case static exclusive cache partitioning would ensure compositionality, but the cache utilization is likely to decrease, because during a scenario only a part of the application's tasks are active, i.e. only a part of the cache is accessed. Thus we introduce a run-time repartitioning method that utilizes information gathered at design-time. Furthermore this method minimizes the penalty involved in cache repartitioning. In our experiments we found that, for realistic scenario switching frequencies, relative to the application number of misses, the inter-task cache flushes are below 4% for the repartitioned cache, whereas for the shared cache it reaches 81%. Moreover, when the L2 is repartitioned according to our method, the relative variations of critical tasks execution time are less than 0.1%, over the entire scenario switching frequency range studied. Regarding the performance, the dynamic repartitioning reduces the number of cache misses per instruction with 33% on average, when compared with the shared cache, resulting in a 10% decrease of the average number of cycles per instruction. Furthermore, when compared with a statically partitioned cache, the dynamic repartitioned one reduces the number of cache misses per instruction with 19% on average, and decreases the number of cycles per instruction with 3% on average.

- Finally, Chapter 7 wraps up the dissertation, by presenting our conclusions and indicating significant and promising follow-up research directions.

Chapter 2

Background

As mentioned in the introduction our work focuses primarily on multimedia applications executing on multi-processors with cache hierarchies. In this chapter we first define what we consider as being a multimedia application, and the features the such application should have (namely predictability, flexibility, and compositionality), as argued in the introduction. Second we introduce the targeted hardware platform, i.e., the CAKE (Computer Architecture for Killer Experience) template. Then we briefly present the benchmark software programs used in the rest of this thesis, and finally we sumarize this chapter.

2.1 Preliminaries

From the entire range of embedded systems, we target only the multimedia embedded systems, i.e., embedded systems that combine processing of text, graphics, video, and sound. We consider that a multimedia system consists of a collection of software tasks addressed here as the application, that execute on a hardware platform constituted by a set of processing resources and a set of memory elements. As tasks do not execute all the time, due to cost reasons, the number of resources is smaller than the number of tasks of the application, i.e., tasks share resources.

Multimedia systems have to process a certain amount of data before a time deadline. Thus a correct execution, implies not only a non-faulty functional execution, but also that the application timing is according to the specifications, implying that the application behavior has to be *predictable*. A system

is considered to be fully predictable if its exact performance can be foretold in every possible situation the systems might encounter. The systems that require full predictability are systems with hard deadlines, i.e., hard real-time systems. Example of hard real time systems are breaking system of cars, safety systems, etc. While designing such systems, every possible utilization scenario and environment situation should be analyzed in detail. The platform has to be dimensioned in such a way that it can deliver the performance needed for the worst case scenario.

When looking at the memory hierarchy, full predictability implies that the latency of each data access should have a known bound. In general, to determine the latency of an access one has to know on which hierarchy level the data reside. Hence, the location of each datum should be known in every clock cycle, thus explicit, detailed traffic regulation is required. As already mentioned, we consider applications consisting of concurrent tasks. To be able to enforce a strict data traffic control, each instruction inside of a task has to be analyzed for each possible combination of tasks. The resulting system is rather rigid, in the sense that all this analysis has to be performed again if one of the tasks is slightly changing its memory demands, or it is updated to accommodate for extra functionality.

As already explained in the introduction, *flexibility* is another very important property of embedded systems. We define flexibility as the ability of a system to accommodate changes and to reuse already designed modules. When a design reuses modules of other existing systems the total engineering cost and effort decreases, leading to a faster time to market. Moreover, the costs can further diminish when a system supports on the fly updates to new emerging features or standards. Thus flexibility offers premises for a successful product as it enables faster time to market, and lowers production costs.

Ideally, flexibility should be present in both the computation and the memory parts of a platform. For the memory this means that data traffic should be automatically routed through the hierarchy levels, such that application changes do not trigger an entire memory hierarchy redesign. Caches represent such a flexible memory organization, nevertheless notorious for their unpredictability [13], [95]. As we can see, in the case of memories, the flexibility clashes with the predictability, as unlike caches, fully predictable memory organizations are rigid. In the following we discuss in detail the caches predictability problem.

In [5], [16], [34], [91], [119] the authors propose methods to enable the prediction of caches behaviour such that they can be utilized for isolated soft

real-time tasks. Thus in the single task context the predictability of the memory hierarchy issue can be considered as solved. However, predicting the performance of a parallel multimedia system involves predicting the performance of the sequential tasks as well as the amount of interference among them. The inter-task interference is caused by resource sharing. Each platform's resource has a certain capacity, e.g., size for memories. A resource can be shared in two manners: (1) multiple tasks use the entire resource one by one, in a time multiplexed fashion (the typical case for processors, busses), or (2) each task uses a fraction of the resource capacity, for the entire task execution time (the typical case for memories). Hence predicting the inter-task interference requires detailed knowledge about all possible task's execution order, timing, and fractions of used resources. In the systems build today such detailed knowledge is practically impossible to acquire, because the number of tasks is large and the resources are complex. Hence, ideally, the performance of each individual sequential part must be preserved if the parts are executed concurrently in arbitrary combinations or if additional parts are added. A system satisfying this property is addressed as being *compositional*.

Subsequently to the two ways in which tasks can share a resource, the compositionality can be spatial or temporal (i.e., the meaning of the word "performance" in the above definition is capacity or time related) [35]. When a resource is shared in a time multiplexed fashion, compositionality reflects the fact that the time quantum a task is served by the resource should be independent of other tasks (temporal compositionality). For example, for a processor, compositionality implies an un-disrupted task execution time (performance is given by the execution time). When a task utilizes a fraction from a resource, the compositionality implies that the size of that fraction should be independent of other tasks (spacial compositionality). For example, for a memory, compositionality implies each task can un-disruptively utilize an guaranteed part of the memory (performance is given by utilized bandwidth). Because in this thesis we concentrate on the memory hierarchy, we focus on the spatial compositionality shortly addressed in the rest of this dissertation as compositionality, unless explicitly specified otherwise. In the end the most important for real-time systems is that the time in which an application produces an output datum is guaranteed. However, the size of the fraction a task uses from a resource has a large influence on its timing, thus spacial compositionality is crucial. Consequently, we believe that in future parallel systems acceptable levels of flexibility and predictability would be very difficult to achieve simultaneously in the absence of compositionality.

Nevertheless, full predictability is not required for many media applica-

tions for which the users can accommodate some degree of variation in the output quality (for instance, a human eye barely notices if on a video screen sometimes, instead of 30 frames per second, we display only 29 of the original frames, and one is displayed twice). Often for these applications the average case resource demands is significantly lower than the worst case [52]. Moreover, the worst case may rarely occur, hence many resources may be waisted for most of the time if the platform is provisioned for the worst. Additionally, media algorithms are complex and data dependent, therefore sometimes it is even impossible to prognosticate every conceivable situation they might be confronted to. Hence, full predictability might be too restrictive and expensive for some multimedia platforms. In view of this, for the case of multimedia applications, it might be economically viable to equip the platform for the average case, and to provide a fall-back mechanism for the situations when the resources do not suffice. Such a system is referred in the literature as ”soft real-time”, and its performance can be foretold only for a limited number of (preferably relevant) average case input data streams, thus it has a limited predictability. In this thesis we target soft real-time system, and for ease of reading, in we simply use ”predictable” for systems with limited predictability, unless explicitly mentioned otherwise.

In the following section we introduce the targeted hardware platform and its memory hierarchy organization.

2.2 The CAKE multi-processor architecture

A media platform, like every computation platform, consists of a set of processing units (for performing calculations among operands) and of a memory subsystem (for operand storing and retrieving). As mentioned in the previous section, we concentrate on applications consisting of a set of software tasks. Our approach does not decline the use of dedicated hardware for some parts of the platform. The proposed method is applicable to them as well, if those parts access the main memory subsystem, together with the software tasks. Otherwise, if the dedicated hardware has its own exclusive memory part, its management is outside of our concern.

In this section we detail the targeted hardware platform, first discussing issues related to the processing part and then to the memory part.

2.2.1 Parallel processing on CAKE

Typically multimedia applications exhibit various levels of parallelism. Without claiming to make an exhaustive study, for clarity purposes, we categorize these parallelism levels as follows. First level, the coarsest, is the *application level*. In our view, an application consists of a collection of tasks implementing the functionality of a separate part of a system or of an entire system. Applications are independent of each-other, or they communicate very little, every one of them providing the functionality of an entire system. For example, we can consider that, in an automotive electronic system, the navigation system and the traction control system are different applications. Large systems may consist of multiple applications executed in parallel. At its turn, an application can be split into parts, called tasks. Thus, the second parallelism level is the *task level*. Such tasks might communicate data and/or share instructions with other tasks For instance, the mentioned car navigation system may consists of the wireless communication task that sends the positioning data to the map rendering task. A tasks is composed by a set of instructions, thus the third, and the finest granularity level, is the *instruction level* (for example on a pipelined processor, multiple instructions may execute simultaneously). In the context of this research we do not quantitatively define the size of the applications, tasks, and instructions, nor the amount of communication involved among them. It is very likely that these sizes vary from one system to another, depending on the functionality and the complexity of the involved algorithms. For example, in a simple music player device the audio decoding might represent an application whereas, in a complex TV system the audio decoding is just a task, next to other video an text processing tasks. The purpose of the present categorization is to have a structural view of a potential system. In summary, an application might expose various levels of parallelism: from application to instruction level.

The envisaged architecture supports parallel execution at all previously mentioned levels. On a top view, the platform consists of a homogeneous network of computing tiles on a single chip [114], as graphically depicted in Figure 2.1. Tiles are connected by a network, for example a torus one, as Figure 2.1 suggests. Each tile may comprise several CPUs (cores like: TriMedia [82], MIPS [65], etc.), hardware accelerators (image enhancement IPs, dedicated video decoders, etc.), a router (for out-tile communication), I/O interfaces, and memory banks, as described in Figure 2.2. The tiled organization enables the exploitation of a coarse grain parallelism. Because inter-tile communication is expensive in terms of time, tiles are useful when the ”entities”

that have to be executed simultaneously do not exchange too much data or do not synchronize very often. According to our prior categorization, these entities are the applications.

On a lower level, each tile contains several CPUs that can execute several tasks in the same time, therefore offering the premises for exploiting task level parallelism. Moreover, each processor core may have a number of functional units that can execute in parallel multiple instructions of the same task. For instance TriMedia is a Very Long Instruction Word (VLIW) processor [82], comprising several issue slots that can execute up to 5 instructions in parallel. In conclusion, the CAKE platform is a very flexible one, offering parallelism at every possible level, from application to instruction.

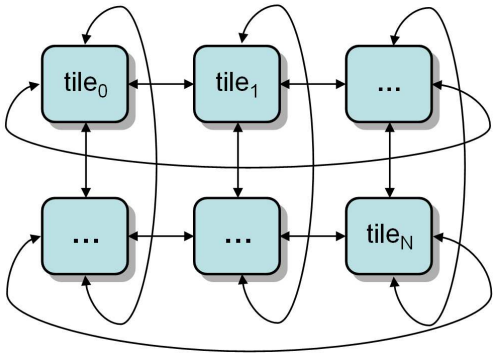

Figure 2.1: CAKE multi-tile architecture (overview).

2.2.2 Memory organization

On every computation platform the latency of a memory access is crucial for the system performance. Ideally, in order to allow a processor to perform the maximum possible number of operations, the operands should be immediately available in each cycle. If the operands are not available, the processor has to stall, waiting for the memory to provide the desired data.

Typically in the CAKE organization, each CPU core inside a tile may have a level one (L1) cache. This cache level is the first to be accessed when the processor needs new instructions or operands that are not present in its registers. Therefore, to achieve the maximum possible theoretical performance, an L1 cache should be as fast as the processor and should always contain the desired data. For nowadays technologies, the latency of accessing a memory location

varies inversely with the memory size [45]. Hence, to achieve the desired performance, cache designers have to mitigate two conflicting trends: (1) as applications include more and more functionality, the amount of accessed data tends to increase thus a cache that can store all data should be large, and (2) large caches have a larger access latency than small caches. As a result of this trade-off, in current technologies the L1 caches that are typically used are relatively small. Few examples of the total (data and instructions) cache size for current multimedia processors may be: the ADRES processor from IMEC has 160 KB cache [41], the TriMedia processor from NXP has up to 96KB [82], and the Texas Instrument TMS320D - 112KB cache [110]. However, unfortunately, such a small cache might not be able to store all the operands required when executing an application. For example, one single frame of a high definition video stream might have up to 2 MBytes, which obviously does not entirely fit in a state-of-the-art L1 cache. In addition, one can observe an increasing trend in the amount of data that have to be processed in a time unit, therefore it is likely that the number of L1 misses of future applications will experience a severe growth. For every miss that occurs, a slow access to the main memory has to be initiated. Thus, to "capture" as much data as possible close to the processor, media platforms employ more levels of cache memory [102], [29], [110] organized in a hierarchical structure. In our situation, besides the L1 caches, a tile is featured with a shared L2 cache that serve the local tile accesses, such that the off-chip and/or the inter-tile long latency accesses are seldom.

Given that a CAKE tile embeds multiple CPUs, each having its own L1 cache, the problem of cache coherency [38] has to be solved. Coherency troubles arise when multiple processors have to access the same memory location, X, hence X might end up by being present in several L1 caches. If the value stored at X by a processor is not somehow synchronized with the values of X that the other processors see in their L1, some processors may use an out-dated value of X. If the software programmer does not take this effect into account, her/his programs may contain unexpected errors. Avoiding such errors is difficult, as conventional caches are hardware controlled, therefore unaccessible for higher software levels.

Typically, in multiprocessors that comprise multiple caches the synchronization when accessing a shared location is done via a coherence protocol. The CAKE platform offers a hardware coherence protocol that ease the software designer from the burden of keeping the content of the caches up to date. The implemented coherence protocol is an MSI one [101], with a snooping strategy. This implies that every L1 line is marked with a flag indicating if the data cached there have the latest value. The coherence protocol utilizes a

shadow tag directory [109] that is stored close to the L2. If a processor requests a data item that is not present in its L1, or it does not have the latest value, the coherence protocol initiates a so-called snoop operation. This operation is actually an inquiry of all the other (on-tile) L1s and of the shared L2, to detect the location of the up-to-date value and to set the cache coherency flags accordingly. In the case of an L2 miss, the cache is refilled with a new data block from the off-chip memory, thus the shadow tags have to be synchronized with the tag of the data newly brought in the L2. the data refill from the off-chip memory. The result is that, transparent to the application, coherent data sharing among processors is enabled. Moreover, because the storage space on a CAKE tile is provided by a hardware controlled cache, the application programmer does not have to be concerned with the underlying cache coherency model. Thus, assuming an application described in a high-level programming language, one may very easily exercise it on a CAKE instance (given, off-course, that a compiler is available to transform the high-level description in executable code for the processors embedded on the platform). This flexibility exhibited by the CAKE platform allows for early stage performance investigation and design exploration. These are very important features when having to achieve a fast time to market.

In general, tasks executing on multiprocessors share the available platform resources, in order to utilize the hardware as much as possible. When several tasks need the same resource simultaneously, inter-task contentions might occur. These contentions are unpredictable and they are causing dependencies among tasks performance, therefore the system is not compositional. Every shared resource on the platform, like caches, busses, memory interface, etc., is subject to contention, and, it has to be managed for compositionality. In this context, in the last years NXP invested a lot of research effort into multiprocessor platform management at different levels [2] [115]. From the large topic of multiprocessor platform management this dissertation deals with multiprocessor caches hierarchy management. The following section present details about the CAKE platform instance that we utilize to experiment our ideas.

2.2.3 The experimental CAKE instance

We consider that an application is running on an instance of the CAKE processor consisting of a single tile detailed in Figure 2.2. This practical tile instance contains a collection of $\mathcal{P} = \{P_k\}_{(k=1,O)}$ homogeneous media processor cores (in our case TriMedia cores [82]), and one control processor core (in our case MIPS cores [65]). The media cores are used for data processing, whereas the

control processor is used for platform management activities (like for example, as we propose in this thesis, for the cache management). Each of the tile's core has its own instruction and data L1 caches, and we assume the existence of a cache coherence protocol as described in the previous section.

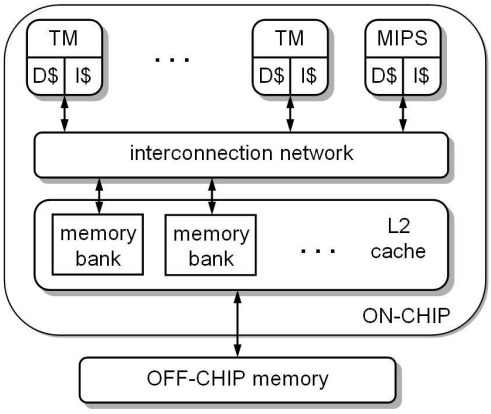

Figure 2.2: Multi-processor target architecture

The L1 instruction cache of each Trimedia core has 32 Kilo Bytes (KB), it is 8 ways associative and has a line size of 64 Bytes. The data cache of a Trimedia core has 16 KB, it is 8 ways associative, and has a line size of 64 Bytes. The latency of accessing the L1 cache is 2 cycles. The MIPS core has an instruction L1 of 16 KB, 8 ways associative, and with a line size of 64 bytes. The data cache of the MIPS is 8 ways associative, has a size of 16KB and its lines contain 64 bytes each. The L2 cache has a line size of 512 Bytes and the latency of accessing such a line is 10 processor cycles (note that this an approximate figure, as it also contain the interconnection network latency, which may vary if not explicitly managed). In our experiments we investigate the behavior of such an L2 for different sizes and associativities. Further we consider 256 Mega Bytes (MB) off-chip memory, that is accessible per blocks of 256 or 512 Bytes. The access latency to off-chip memory is up to 100 processor cycles for the first word of a block and 1 cycle for each subsequent word in the same block (here the same considerations as for the L2's latency are valid, namely that variations may exist when not explicitly managed).

2.3 Application model

The set of all applications exercised on a CAKE platform is denoted with $\mathcal{A} = \{A_l\}_{(l=1,B)}$. We assume that an application A_l consist of a set of N_l software tasks, $\mathcal{T}_l = \{T_i\}_{(i=1,N_l)}$. As we do not discuss more than an application at once, in the rest of this dissertation the index l is skipped, thus an application is denoted just with A, its task set with $\mathcal{T}$.

In the first part of this thesis we consider applications as being static in the sense that tasks do not stop or resume. However, in Chapter 6 we discuss applications that have multiple utilization scenarios, meaning that not all the tasks are continuously active. For instance, in a personal digital assistant device the audio decoding task is active only when the user listens to music. Thus, tasks may start and stop, depending on the user requests. In this case an application has a set $S = \{S_q\}_{(q=1,2,...,U)}$ of possible scenarios. In each scenario S_q only a subset of tasks $\mathcal{T}_q \subseteq \mathcal{T}$ is active. Note that a static application can be assimilated with an application with a single execution scenario in with all $\mathcal{T}$ tasks are active.

Four types of parallelism are possible among the tasks of an application: (1) no-dependency parallelism (tasks are perfectly independent and they actually implement separated functionality), (2) functional parallelism (tasks perform different operations in a pipeline manner, one task producing the input of another), (3) data parallelism (tasks perform the same operation on different parts of the input data), and (4) a mix of the previous three types.

In case the application exhibits data parallelism, multiple tasks execute the same instructions on different parts of the input data, so they share instructions. When functional parallelism exists, the tasks can exchange data (a classical example of shared data are the reference frames of video decoder or/and encoders). Thus, in media applications tasks may share instructions and/or data. As there is no principle difference between sharing data and instructions, for simplicity we use in the remainder of this thesis the term ”common regions” for both inter-task shared data and instructions. We consider that an application A has M common regions, denoted with $\mathcal{CR} = \{CR_j\}_{(j=1,M)}$. In case the tasks comprising the application are independent, no common region exists.

Various application programming models are proposed in the literature. Among the ones that are appropriate for media processing we mention: streaming models, series-parallel models, finite state machine models, etc. The CAKE platform supports among others the following models: Pthreads [14], Y-chart Application Programming Interface (YAPI) [25], and Task Transac-

tion Level (TTL) [113]. As one can see, the application model we describe in this chapter is general enough to support each programming model that has a task as the basic organization unit, like most of the programming models on CAKE. For the practical experiments we use applications that exhibit all of the mentioned parallelism types. These applications are mainly described in YAPI (the last part of this section details the circumstances when that's not the case).

The computation model in YAPI is based on Kahn Process Networks. Such a network consists of parallel tasks that communicate through (theoretically unbounded) FIFOs. YAPI offers primitives for creating tasks and for linking them using FIFO channels, forming the so-called networks (applications). In practical implementations the FIFOs are bounded and blocking. A task blocks if it reads from an empty FIFO or if it writes in a full FIFO. The advantage of this operation mode is that the application programmer does not have to worry about task synchronization at data access.

However, YAPI communication model is not efficient when tasks have to exchange large amounts of data through FIFOs. This is because every data exchange implies two memory copying operations: one when the producer writes into the FIFO, and another one when the consumer reads the data. For large data items, like video frame buffers, copying the data twice incurs a large penalty. In such cases only one data copy is stored, and all the tasks are accessing it. The synchronization at data access has to be explicit and may be performed by every classic primitive like barriers, semaphores, etc. In this case, we use blocking FIFOs to send synchronization tokens that indicate data availability.

In general, there are multiple possible ways to exchange data among tasks when necessary. We take into consideration only the ones that are based on memory buffer sharing, such that tasks communicate through the memory hierarchy, thus through the shared L2. When memory parts are shared by multiple tasks, synchronization is necessary to ensure that data are not read before written (data correctness). There are two types of relations among data exchange and data synchronization: (1) data synchronization is directly combined with data exchange; blocking FIFOs, for example, fall in this category; in this case, inter-task synchronization is transparent to the application programmer. (2) synchronization is separate from data exchange. In this case, inter-task synchronization has to be explicitly taken into account by the application programmer. Both ways are supported by the CAKE platform as the CAKE platform has a light weighted Operating System (OS) that besides scheduling and mapping offers synchronization primitives (critical regions, semaphores,

barriers [108]), and dynamic memory allocation/deallocation functions.

In the following section we present the manner in which the application's software tasks are mapped on a CAKE multiprocessor.

2.4 Tasks allocation and scheduling

As specified in the previous section, the cache management method deals with software tasks which are executed on hardware programmable processors. On a multiprocessor platform, an OS is typically the one determining when a task starts and/or stops and on which processor it runs (scheduling and allocation). In the literature multiple operating system scheduling and mapping policies have been reported. For an overview of the real-time related ones we refer the reader to [64]. In our case, because media applications are data intensive, we regard a task generically, as a process consuming input data and producing output data. Tasks are (naturally) synchronized based on data availability. Therefore, in the considered setup a task temporarily stops its execution (it is swapped out) in two cases: (1) when task's input data buffers are empty or its output buffers are full (data availability), (2) when the OS decides that other tasks has to restart its execution. Between two executions of the same task, a processor can execute other tasks. In any case, in order to support a natural load balancing, the tasks may freely migrate from one processor to another, depending on the processors availability.

In the following section we briefly introduce the benchmark applications used for the experimental part of our work.

2.5 Benchmark applications

As previously mentioned, we investigate two main types of applications: static and dynamic. Furthermore, we exercise two subtypes of static applications, the first consisting of communicating tasks and the second consisting of non-communicating tasks. The dynamic applications consist of non-communicating tasks, and we build several experimental scenarios as presented in Subsection 2.5.2. The applications consisting of communicating tasks that are available for the experiments cannot be exercised for dynamic repartitioning, because all the tasks of such an application are working together for the same purpose (video decoding) and they are not prone to scenarios that start and stop.

2.5.1 Applications with communicating tasks

We use two large, industrial relevant applications composed out of communicating tasks: a picture in picture TV (PiPTV) based on the work in [25], and an H.264 decoder [26], [50].

The PiPTV consists of two main decoding streams, each of them comprising a scaling task, a video multiplexing task, an mpeg2 decoder and a video demultiplexing, as depicted in Figure 2.3. The mpeg2 decoder is actually a set of several tasks (omitted from Figure 2.3 for clarity reasons): idct, quantization, and motion estimation. The PiPTV application exhibits functional parallelism and the data are exchanged through FIFOs (represented as edges in the graphs in Figure 2.3). The tasks actually come in pairs corresponding to the two different picture streams, therefore they share instructions.

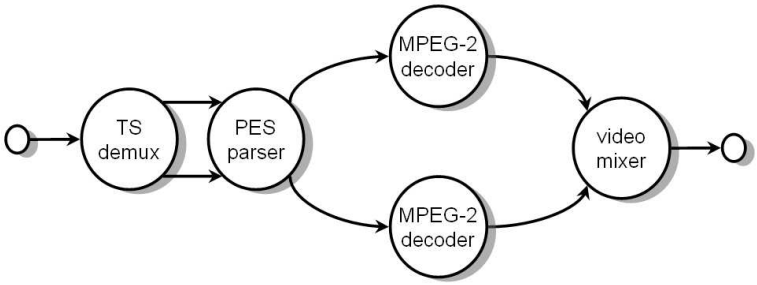

Figure 2.3: PiPTV parallel decoder

The H.264 decoder application is formed by several tasks, and it is schematically presented in Figure 2.4 (due to clarity reasons not all FIFOs are depicted). First an entropy decoder task processes the input stream and passes the data via a scheduler to a set of transform decoder and loop filter tasks. The transform decoders and loop filters execute inverse quantization, transformation, prediction, and deblocking, respectively. The H.264 exhibits functional parallelism among the entropy encoder, the scheduler, the transform decoders, and the loopfilters. Moreover among each transform decoder and loop filter, respectively, there is data parallelism as these tasks process different parts of the image. Thus these two types of tasks share their instructions. The video frames are also shared and the synchronization is performed through FIFOs.

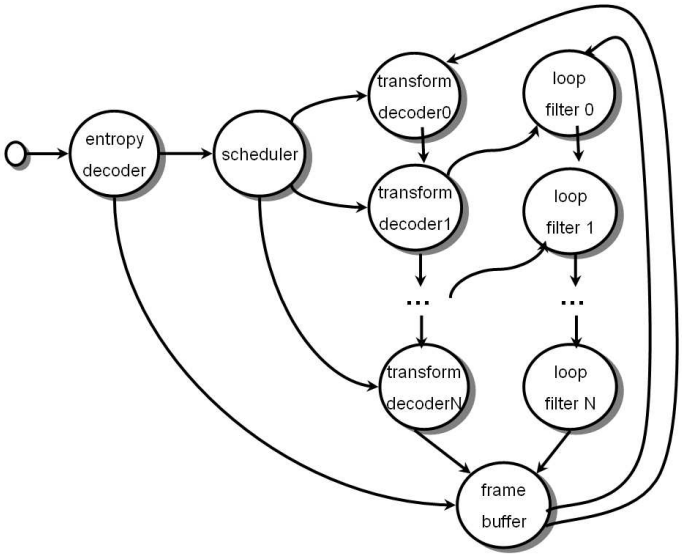

Figure 2.4: H.264 parallel decoder

2.5.2 Applications with non-communicating tasks

In order to enlarge our experimental spectrum we also map on CAKE several applications consisting of non-communicating tasks. We use various multimedia programs from the MediaBench benchmark [22]. From this collection of programs we pruned out the ones that are relatively small and not memory intensive. Moreover, in order to make the benchmark more representative for emerging technologies, we augmented the MediaBench suite with an H.264 video processing program. In the experimental framework, an application is formed by a collection of four such programs, each of them representing now a task. Table 2.1 presents the set of 10 programs, each of them becoming a task in an exercised application. All of these are reasonably memory intensive workloads. Table 2.2 presents the tasks that compose each of the 6 applications.

Furthermore, to exercise the dynamic cache management we build 7 execution scenarios (chosen at random from the total set of possible tasks combinations) for each of the 6 applications, as presented in Table 2.3. As some tasks are more sensitive perturbations than others, we denote the sensitive ones as critical and we present them in bold in Table 2.3. These critical tasks should not be disturbed at scenario switch, when other tasks start and stop.

H.264 encoder	A very low bit-rate video encoder (h264enc) based on the H.264 standard.
H.264 decoder	A very low bit-rate video decoder (h264dec) based on the H.264 standard.
MPEG2 encoder	A motion video compression encoder (mpeg2enc) for high-quality video transmission, based on the MPEG-2 standard.
MPEG2 decoder	A motion video compression decoder (mpeg2dec) for high-quality video transmission, based on the MPEG-2 standard.
EPIC encoder	An image compression coder (epic) based on wavelets and including run-length/Huffman entropy coding.
EPIC decoder	An image compression decoder (unepic) based on wavelets and including run-length/Huffman entropy coding.
Audio encoder	MPEG-1 Layer III (MP3) audio encoder.
Audio decoder	MPEG-1 Layer III (MP3) audio decoder.
JPEG encoder	A lossy image compression encoder for color and gray-scale images, based on the JPEG standard.
JPEG decoder	A lossy image compression decoder for color and gray-scale images, based on the JPEG standard.

Table 2.1: Used media tasks

A_1	JPEG encoder (JPEGe); JPEG decoder (JPEGd); H.264 encoder (H264e); Audio encoder (AUDe);
A_2	H.264 decoder (H264d); MPEG2 decoder (MPG2d); EPIC decoder (EPICd); Audio decoder (AUDd);
A_3	JPEG encoder (JPEGe; Audio encoder (AUDe); Audio decoder (AUDd); MPEG2 encoder (MPG2e);
A_4	H.264 encoder (H264e); MPEG2 encoder (MPG2e); EPIC decoder; JPEG decoder (JPEGd);
A_5	MPEG2 decoder (MPG2d); Audio encoder (AUDe); JPEG decoder (JPEGd); EPIC decoder (EPICd);
A_6	H.264 decoder (H264d); Audio decoder (AUDd); JPEG encoder (JPEGe); MPEG2 encoder (MPG2e);

Table 2.2: Applications and their tasks

	S_1	S_2	S_3	S_4	S_5	S_6	S_7
A_1	**JPEGe**	**JPEGe**	**JPEGe**	**JPEGe**	**JPEGd**	H264e	H264e
	JPEGd	JPEGd	JPEGd	H264e	H264e	**AUDe**	-
	H264e	H264e	**AUDe**	**AUDe**	**AUDe**	-	-
	AUDe	-	-	-	-	-	-
A_2	H264d	H264d	EPICd	H264d	EPICd	H264d	H264d
	MPG2d	MPG2d	MPG2d	**EPICd**	MPG2d	**EPICd**	MPG2d
	EPICd	**EPICd**	**AUDd**	**AUDd**	-	-	-
	AUDd	-	-	-	-	-	-
A_3	**MPG2e**	**MPG2e**	**MPG2e**	**MPG2e**	**MPG2e**	**MPG2e**	**MPG2e**
	AUDe	AUDe	AUDd	JPEGe	AUDe	AUDd	-
	AUDd	AUDd 32/8	JPEGe	AUDe	-	-	-
	JPEGe	-	-	-	-	-	-
A_4	**H264e**	**H264e**	MPG2e	**H264e**	**H264e**	JPEGd	**H264e**
	MPG2e	EPICd	EPICd	EPICd	EPICd	EPICd	-
	EPICd	JPEGd	JPEGd	MPG2e	-	-	-
	JPEGd	-	-	-	-	-	-
A_5	**EPICd**	**EPICd**	**EPICd**	**EPICd**	**EPICd**	MPG2d	AUDd
	MPG2d	AUDd	MPG2d	JPEGd	AUDd	AUDd	JPEGd
	AUDd	MPG2d	JPEGd	AUDd	-	-	-
	JPEGd	-	-	-	-	-	-
A_6	**H264d**	**H264d**	**H264d**	**H264d**	MPG2e 128/u	**H264d**	**JPEGe**
	AUDe	AUDe	AUDe	AUDe	AUDe	MPG2e	-
	JPEGe	**JPEGe**	MPG2e	**JPEGe**	-	-	-
	MPG2e	-	-	-	-	-	-

Table 2.3: Applications and execution scenarios

2.6 Summary

This chapter first presented the necessary terminology and definitions related to multimedia embedded systems desired properties, namely predictability, flexibility, and compositionality. Then we described the targeted platform (the CAKE multiprocessor) and applications (multimedia, multitasking). The CAKE platform contains multiple processor core, each of which typically having an own level of cache. Moreover the platform contains a large shared level two cache. On this platform one can run application consisting of multiple tasks that may share instructions and/or data. These applications may be static, in the sense that tasks do not start and/or stop at run-time, or dynamic if different execution scenarios (i.e., tasks' start and stop) may occur at run-time. Moreover, we introduced the benchmark application suite utilized in the experiments in the rest of this thesis. These applications may consist of independent tasks from MediaBench augmented with and H264 codec, or of communicating tasks (H.264 and PiPTV).

The next chapter introduces two potential compositional cache organizations and investigates their performance.

Chapter 3

Cache partitioning

In Chapter 1 we briefly reviewed the options for on-chip memory organization. We identified two major types of data traffic control thorough the memory hierarchy. The on/off chip traffic can be explicitly controlled by the programmer or the compiler (scratch-pad) or implicitly controlled at run-time (cache). Both these options have their advantages and shortcomings, scratch-pad is predictable but not flexible, whereas the cache is flexible but unpredictable. In view of the discussion in Chapter 1, we propose to use an implicitly controlled scheme (for flexibility reasons), but to manage it in such a way that its predictability increases. The management consists in restricting a task's access to the cache memory such that the task can only utilize an exclusive region of it. This means that if a task T_i needs its data on chip, the traffic control mechanism cannot swap another's task T_j data, but it has to swap some other part of T_i's data. In this way the amount of data swapping depends on the memory size allocated to a task, and doesn't depend on other tasks behavior. This task separation induces a *compositional* cache access. Because tasks don't influence each other memory performance, no detailed analysis of the entire application is needed when a software task is updated. Instead, a local analysis of the changed task can be performed to determine its new cache requirements. As we see, our proposal slightly restricts the full flexibility of caches, but improves predictability (due to compositional tasks performance). In this chapter we analyze the possibilities for enforcing such a task centric management on an implicitly controlled memory.

The remainder of this chapter is organized as follows. In Section 3.1 we discuss the conventional organization of the implicit controlled memory (cache) and in Section 3.2 we investigate the cache partitioning options avail-

able for the envisaged task centric management scheme. The software support needed for cache partitioning is presented in Section 3.3. Then Section 3.4 describes a method to determine the partitioning ratio such that the total application number of misses is minimized. This is followed by Section 3.5 where we introduce a metric to quantitatively evaluate the compositionality property. In the final part of this chapter, Section 3.6 presents an experimental comparison of compositionality and performance, among the possible cache partitioning schemes and the conventional shared cache. Finally, in Section 3.7 we draw the conclusions of this chapter.

3.1 Conventional cache organization

We consider a cache memory as being logically organized as a rectangular array of memory elements arranged in ”sets” (rows) and ”ways” (columns) [38], like in Figure 3.1. The number of ways in a set is denoted as the cache's ”associativity”. The information stored in each way consists of: (1) a few control bits (*C*), (2) tag bits which are part of the address of the cached data (*TAG*), and (3) the actual data bits (*DATA*), also called ”cache line”, containing more data words. The associativity of a cache is a design option and the cache organization can vary from the scenario in which every set has one single way (direct mapped cache), to the scenario in which there is only one set in the cache, each line representing a way (fully associative cache).

Subsequently to the cache organization, an accessed address is logically split in three fields: *tag*, *index*, and *offset*. The *offset* part of the address identifies the required data word inside a cache line. The set where a data item can be placed is uniquely identified by the *index* part of the address (i.e., the mapping between an address and a cache set is done via a modulo function). Inside that set, the data may reside in one of the ways. In case some data item is required, all the ways are searched to determine if and in which way it is cached. The data look-up is done by comparing the *tag* address part with the tag bits stored in each way of the cache in the corresponding set. If the requested data item is present in the cache we have a so called ”cache hit”. If the data item is not found in cache (so called ”cache miss”), it is loaded by the cache controller. Loading of new data in a cache set implies that, if the set is full, some of the data already located there have to be swapped out of the cache (action also known in the literature as ”cache line victimizing”). The way flushed out of the cache is decided by the replacement policy. Multiple options for replacement policies exist: last recently used (LRU), first in first

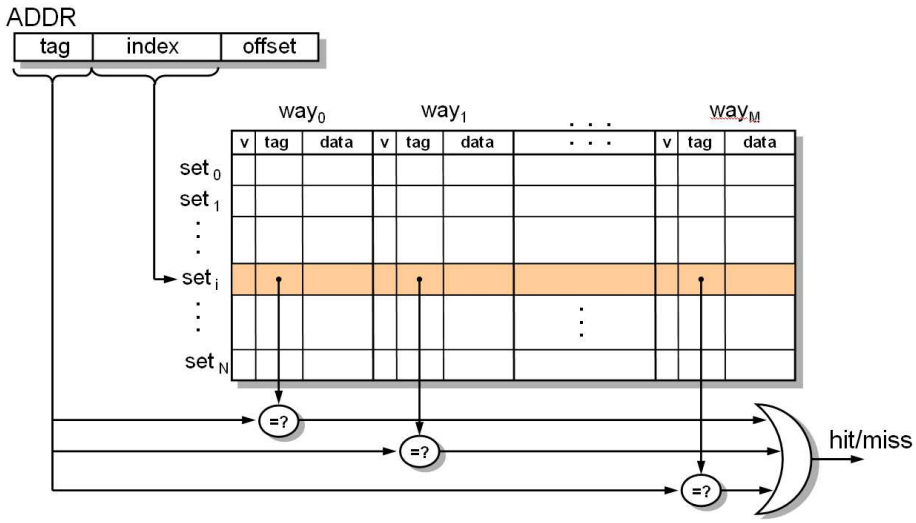

Figure 3.1: Conventional cache organization.

out (FIFO), random, etc, [38]. In the rest of this thesis we refer to caches as having the logical organization presented in this section, while the replacement policy may be any of the mentioned one.

The presented cache organization is meant to exploit two types of program locality: spacial and temporal. The spacial locality refers to the fact that programs tend to access multiple consecutive addresses. This is the reason why bringing new data in cache is performed per cache line, so if consecutive words are accessed only the first in the block causes a miss. The temporal locality refers to the fact that the same address might be accessed multiple times by the program. In this case, if data are not swapped out among the two consecutive accesses at the same address, the second accesses will surely result in a cache hit.

The cache misses that might occur in a cache are classified in three types: (1) compulsory misses (they occur at the first reference of a datum), capacity misses (are the misses that occur in a cache of a certain size, regardless of its associativity or line dimension), and conflict misses (the reasons of their existence is the fact that the associativity of the cache is not large enough, therefore many addresses are mapped on the same cache line, causing conflicts).

In a traditional cache, neither the index addressing, nor the replacement policy are aware of the internal, task-based, structure of the executed appli-

cation. This unawareness may cause unpredictable inter-task misses which should be avoided in order to ensure compositionality. Our work targets this cache contention, therefore we focus on isolating tasks (assign a cache part to each one of them), such that their number of misses are independent of each other. For this we utilize a cache partitioning scheme. Based on the conventional cache organization, there are two natural partitioning manners: (1) based on associativity and (2) based on sets. In the following section we describe in detail these two options and their potential implementation in the context of the CAKE multiprocessor architecture.

3.2 Cache partitioning options

3.2.1 Associativity based partitioning

The associativity based partitioning scheme is depicted in Figure 3.2. As one can observe, each and every task gets a number of ways from every set of the cache. In case the required data item is present in the cache, it is accessed, just like in a conventional cache. However, in case of a miss, when a cache line has to be replaced, one task can flush out only its own cache ways. In this manner, different tasks do not interfere unpredictably.

This type of partitioning is implemented by changing the cache replacement policy as suggested in [21]. This require a small table that specifies which task owns which cache ways, and some extra logic to restrict the victim lines that are to be flushed. This logic is not on the critical path, as the line to be victimize does not have to be known before the data are actually loaded from a lower memory level. On our CAKE platform, loading an L2 line from the main memory takes at least 100 cycles, thus we can consider that there is no time penalty involved in associativity based partitioning. From the area point of view, all the necessary hardware represents a negligible fraction of the size of a L2 cache. This negligible penalty, together with the fact that the implementation doesn't require modifications in the structure of the cache or in the addressing mode [21], leads to a common use of variations of this partitioning type [92], [104], [103] for the purpose of reducing the number of misses and speeding up the application.

In the context of compositionality, the main shortcoming of associativity based partitioning is that the number of allocable resources is restricted to the number of ways in a set (cache organization). A state-of-the art L2 cache typically has only up to 16 ways. Every extra way present in a cache requires

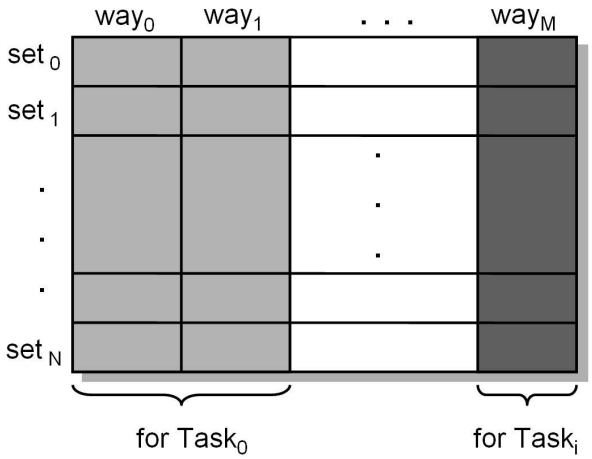

Figure 3.2: Associativity based cache partitioning (logic organization).

and extra comparator [38]. Thus the reason for supporting just few ways is that the extra circuitry involved in implementing associativity increase the cache access time and the power consumption at each lookup. In media applications there is a trend in adding new features, so in increasing the number of tasks. Consequently, for such an application there might not be enough ways for every task, therefore multiple tasks would have to share the same way, leading to unforeseeable cache interference.

3.2.2 Set based partitioning

The set based partitioning scheme is illustrated in Figure 3.3. In this case, each and every task gets a different amount of sets from the cache. As already mentioned, in a conventional set associative cache organization the address splits into three parts: tag, index, and offset. Set based partitioning implies that the addresses a task may access can have only some restricted indexes, pointing to the task's cache sets. This is equivalent with an address space partitioning. To the best of our knowledge, there are two previous approaches to implement this address space partitioning. One implements the partitioning at compiler and linker level [77] and the other at operating system level [63]. In the scheme proposed in [77] the compiler and the linker allocate variables and instructions addresses such that the cache partitioning is achieved. The platforms we consider may contain standard processor cores, thus the compilers are developed by external parties. A platform specific change of the compiler would be costly and/or time consuming, therefore we do not follow this path.

The cache partitioning method controlled by the operating system proposed in [63] has also drawbacks as it is limited to physically indexed caches and requires a virtual memory model. Nevertheless, we would like to support all types of caches on platforms with or without memory paging. Consequently, none of the existing method is suitable for our purpose, thus in the following we propose a new technique to implement the set based cache partitioning.

We achieve the cache partitioning through a level of indirection, without interfering with the memory space. This is somewhat similar with the mechanism in [63], but the address translation is not performed at memory page level, but directly at cache level. In this manner there is no restriction in the type of supported cache, nor in the underlying memory model. Our scheme modifies the index bits of an address into new index bits, before cache lookup, as depicted in Figure 3.4, by taking into account who initiates the access. The purpose of the index translation is to send all the access of a task T_i, and only the accesses of task T_i, in a cache region decided at design-time.

To avoid expensive index calculation, the partition sizes are limited to power of two number of sets. We propose to use a table (indexed by the *task id*) to maintain the information needed for the index translation (*MASK* and *BASE* bits). To clarify the mechanism, let us assume the simple example of an access to data *A*, having the index idx_A in a conventional cache, and belonging to task T_i. We denote by 2^k the size of the partition for T_i and by 2^C the size of the total cache (both size values are measured in sets). The $MASK_i$ bits actually select the k least representative bits of idx_A (i.e., instead of doing modulo with the cache size, 2^C, the modulo is done with the partition size 2^k). $BASE_i$ fills the rest of the C-k index bits such that different tasks accesses are routed in disjoint parts of the cache.

After index translation, two addresses that did not have the same original index might end up having the same new index. In this case the system is not able to distinguish among such two addresses, leading to data corruption. To prevent data corruption, the index bits changed by the translation process still have to identify somehow the associated memory access. The easiest way to achieve this is to augment the tag part of the address with those changed index bits. For our example, task T_i has 2^k cache sets thus the C-k most representative index bits are changed, and have to be included in the tag. Because it is not beneficial to have a tag with variable length (k varies with the task's allocated cache size) we choose to augment the tag with all index bits, even though only a fraction of them is actually changed by the index translation procedure. If,

Figure 3.3: Set based cache partitioning (logic organization).

for instance, a 2 MBytes L2, 8 ways associative, 512 Bytes block size is utilized, the tag has 9 extra bits. This overhead represents less that 0.5% of the total L2 area, so the implied area penalty can be considered negligible.

As described in Chapter 2, in this work we assume a multiprocessor platform with cache coherence among the L1 caches of each processor core. In case a task does not find its data in the corresponding L1, a coherence protocol is executed to determine if the data are located in another processor L1 cache. The coherence protocol utilizes a shadow tag directory [109] that is stored close to the L2 and indexed by the original address' index (as the L1s). Consequently, the index translation for the L2 accesses can be performed in parallel with the search in the shadow tags directory, resulting in no additional delay penalty associated to the extra index translation.

As one could see, the implementation of the set base partitioning is more "intrusive" into the cache organization than the implementation of the associativity based partitioning, in the sense that it requires the alteration of the addressing scheme. However, the advantage of this partitioning type comes form the fact that typically, a cache like the L2 we target, may have thousands of sets and only few ways. The number of resources (cache sets in this case) is large, thus set based partitioning permits every task to have its own exclusive part, hence it is the best candidate for achieving compositionality.

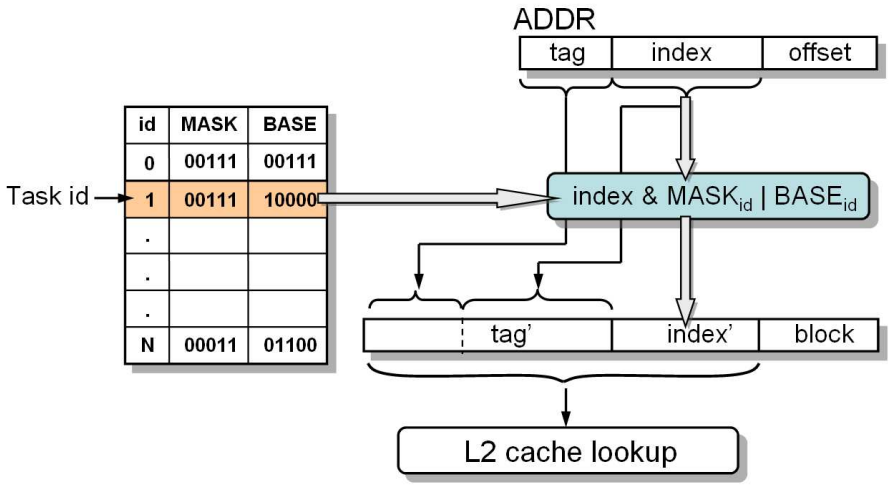

Figure 3.4: Set based cache partitioning (implementation).

3.3 Software support for cache partitioning

In the previous section we described the hardware support needed for cache partitioning. However, this hardware has to be programmed to permit applications to benefit from it. In this section we present the necessary software to control the hardware involved in cache partitioning.

As mentioned in Chapter 2, we assume that our platform has at least a light-weighted operating system responsible for task scheduling. The cache management tables can be programmed via memory mapped I/O. We augment this existing OS with primitives for loading and modifying the necessary cache management tables. Calls to these primitives have to be inserted in the application's initialization code. We mention that the OS task is regarded as a task like any other, therefore it has its own allocated cache part. The cache size that can be allocated to a task is parameterizable. In this manner the application designer has the opportunity to experiment different cache partitioning ratio without having to recompile the entire application.

In the following we discuss the address assignment mechanism and its relation with our task centric cache management. Two issues are of interest: address alignment and dynamic memory allocation.

The basic allocable unit to a task is a cache line, therefore it is not possible

for more than one task to have data in the same line. This implies that, if variables belonging to different tasks have consecutive addresses, these addresses have to be aligned at L2 cache line size. We consider that both the stack and the static data regions of each task are contiguous. As they are known at compile time, it is easy to impose that they are aligned at L2 block size. The dynamic allocated data regions are more interesting, as they cannot be known at compile time, so we discuss them in the next paragraph. We do not comment here the case of shared data among multiple tasks, as we elaborate on this subject in the next chapter.

In some multiprocessor implementations it is possible that the tasks utilize the same contiguous dynamic memory pool (heap), allocating consecutive memory pieces. In this way, a variable allocated and deallocated by a task may get various addresses, depending on how much memory has been allocated by other tasks (memory allocation history). This allocation interleaving can lead to task's variables having diverse addresses in different runs. The cache set where a variable maps is given by a part of the address (index), as mentioned in the beginning of this chapter. As a result, a variable may have different places in cache in different application runs. This is undesirable, because different amount of cache conflicts may occur among a task's variables, depending on the memory amount that other tasks allocate, and the order in which these allocations take place. Thus, the number of misses of a task is not independent of the behavior of other tasks, therefore the system is not compositional.

An option to solve this problem is to impose that the indexes of a task variables do not differ from a run to another. The hierarchical dynamic memory allocation scheme proposed in [98] appears to be a good starting point to implement such a strategy. This scheme is proposed for embedded systems consisting of multiple processing elements and it is implemented in hardware. The original purpose of this scheme is to dynamically allocate memory in a short, deterministic time. We adapt this scheme for our purpose, i.e., to achieve an amount of cache conflicts among a task's variables that is independent of other tasks behavior. Therefore, instead of processing elements, the entities that may allocate memory are task. Similar to [98], when a task allocates memory for the first time, it has to acquire a large, fixed size, memory chunk from the first level of the allocation hierarchy. On a second level of the allocation hierarchy, the task can acquire contiguous memory of desired size from that large memory piece. The addresses of two large memory pieces differ in their most significant bits, therefore it is possible to keep the index part unaltered (the most significant bits of an address belong to the tag part, as described in Section 3.1). The size of such a large memory chunk should be at least equal to

the largest cache size allocated to a task. The advantage of this scheme comes from the fact that the part of the address that differs among these large memory pieces is not used for determining the place inside of the cache part assigned to the task. Therefore, even if some task's variables get different addresses depending on other tasks' allocation history, this does not translate anymore into various places in cache. In this manner the unpredictability is eliminated. Moreover, if the cache line alignment is imposed when allocating the large memory slices, different tasks cannot have data in the same cache line. In the remainder of this thesis we consider that the dynamic memory allocation is performed as described above. Moreover, the fact that the dynamic allocation is performed in hardware or in software is not relevant for our purpose, so any of the two options can be implemented and utilized.

3.4 Cache partitioning ratio

As previously indicated in this chapter, cache partitioning is designed to isolate the tasks in cache therefore to enable compositionality. Orthogonal with the compositionality, cache partitioning offers a degree of freedom in optimizing the application performance (number of misses, throughput, etc.). Given a set of tasks $\mathcal{T}$ and the available cache size C, we identify two optimization problems, formulated as follows:

1. the *The Cache Allocation Problem*, $\mathcal{CAP}$ - determines the cache size (c_i) assigned to each task, such that a certain criterion is optimized. This problem is also denoted as the cache partitioning ratio problem.

2. the *The Cache Mapping Problem*, $\mathcal{CMP}$ - determines the cache line where a task's part begins (b_i), such that a certain criterion is optimized.

Static partitioning methods consider the cache space as being uniform, in the sense that the application performance is influenced only by the tasks cache sizes c_i and not by the beginning cache units b_i. Thus for static partitioning the interesting problem is the *cache allocation*. Therefore in order to exercise the two cache partitioning types, we have to solve the $\mathcal{CAP}$. We revisit $\mathcal{CMP}$ in Chapter 6 where we study the case of applications composed by tasks that might start and stop and we propose a suitable dynamic cache partitioning method.

As mentioned in Chapter 2, in this thesis we assume that an application A is composed out of N tasks, $\mathcal{T} = \{T_i\}_{i=1,2,...,N}$. Each task has assigned a

cache size c_i. We denote the set of possible cache sizes with $\{\sigma_k\}_{k=1,2,...,L}$, thus $c_i \in \{\sigma_k\}$. For the set based cache partitioning, σ_k can be limited to powers of two number of cache sets, due to implementation reasons, whereas for associativity based partitioning no such restriction may apply. Regardless the restriction on σ_k, several optimization criteria can be formulated in relation with $\mathcal{CAP}$, but given that traditionally, the performance of a cache is often measured in number of misses, we formulate the $\mathcal{CAP}$ for finding the size of cache c_i for every task T_i such that the overall number of cache misses is minimized:

$$min\left(\sum_{i=1}^{N} miss(T_i, c_i))\right), \tag{3.1}$$

where, $miss(T_i, c_i)$ is the number of T_i misses when it has c_i amount of cache.

In the following subsections we first prove that the $\mathcal{CAP}$ that minimizes the number of misses is NP-complete, then we present a Dynamic Programming formulation that optimally solves this problem.

3.4.1 Hardness of the cache allocation problem

In this subsection we prove that the cache allocation problem is similar to the Multiple Choice Knapsack Problem ($\mathcal{MCKP}$), that is known to be NP-complete [55].

In order to make this proof we first recall the multiple choice knapsack problem, with the notations from [85]. Given a set of items j ($1 \leq j \leq M$), each of which being of K classes, their profits $\{p_{ij}\}$ and their weights $\{w_{ij}\}$ for each class i ($1 \leq i \leq K$), and a knapsack of capacity c, choose one item from each class such that the total profit is maximized and the total weight does not exceed the knapsack capacity c.

The cache allocation problem is actually a multiple choice knapsack problem, but with different notations, as follows: (1) each class i in $\mathcal{MCKP}$ corresponds to a task T_i in $\mathcal{CAP}$, (2) the knapsack capacity c is the total cache size C, (3) a task may get assigned cache of L possible different sizes which translates in $\mathcal{MCKP}$ in the fact that different items j may be of class i, (4) the weight associated to an item in a class is designated in $\mathcal{CAP}$ by the cache sizes $c_i \in \{\sigma_k\}$ of a task T_i, (5) the profit associated to an item in a class in $\mathcal{MCKP}$ has as correspondent in $\mathcal{CAP}$ the number of misses for each task $miss(T_i, c_i)$. The problem is to pick a single item (cache size) from each class

(task) such that the profit (number of misses) is optimized. The only difference between the two problem formulations is that in $\mathcal{CAP}$ the number of misses is minimized, not maximized as in the case of the profit in $\mathcal{MCKP}$. As this difference is irrelevant for the hardness of the problem, and as $\mathcal{MCKP}$ is known to be NP-hard [55] we can conclude that $\mathcal{CAP}$ is also NP-hard.

A typical manner to solve Knapsack Problems is by utilizing Dynamic Programming. Thus in the following we present a Dynamic Programming formulation that solves the cache allocation problem.

3.4.2 $\mathcal{CAP}$ optimal solution via dynamic programming

From the theory of combinatorial optimization it is known that a problem that satisfies the Bellman's optimality principle can be solved by Dynamic Programming [10]. The optimality principle states that for a problem consisting of making a set of decisions (or choices), in an optimal sequence of decisions/choices, each subsection must also be optimal. For the aforementioned $\mathcal{CAP}$ this would mean that, if a cache to tasks allocation $\{c_1, c_2, ..., c_{N-1}, c_N\}$, with $\sum_{i=1}^{N} c_i \leq C$ is optimal, then a subsequence $\{c_1, c_2, ..., c_{N-1}\}$, with $\sum_{i=1}^{N-1} c_i \leq C - c_N$ would also be optimal. The reason why this holds true can be explained as follows. Let us assume that the cost (number of misses) of the optimal solution $\{c_1, c_2, ..., c_{N-1}, c_N\}$ is $M_N = \sum_{i=1}^{N} m_i = \sum_{i=1}^{N-1} m_i + m_N = M_{N-1} + m_N$, where m_i is the optimal cost for task T_i. If the subsequence $\{c_1, c_2, ..., c_{N-1}\}$ is not optimal, then there exists another allocation $\{c'_1, c'_2, ..., c'_{N-1}\}$ that is optimal for $\sum_{i=1}^{N-1} c_i \leq C - c_N$, therefore the cost of the new allocation M'_{N-1} is smaller then the cost of the original subsequence M_{N-1}. As a consequence, there exists an allocation $\{c'_1, c'_2, ..., c'_{N-1}, c_N\}$ that has a cost $M'_{N-1} + m_N < M_{N-1} + m_N$, but this contradicts with the hypothesis that $\{c_1, c_2, ..., c_{N-1}, c_N\}$ is an optimal solution. Thus $\mathcal{CAP}$ satisfies the optimality principle. This principle is the foundation of the $\mathcal{CAP}$ Dynamic Programming formulation, as it is presented in the rest of this subsection.

Dynamic Programming is a bottom-up approach. The solution is build iteratively by adding a task to the solution from the previous step. The Dynamic Programming formulation of $\mathcal{CAP}$ is described in Algorithm 1. As one can notice Algorithm 1 has N main steps. At the ith step of the algorithm a cache allocation for the first i tasks is found, for every possible cache

size $c \leq$ *C-(N-i)*, such that each task has at least one cache set ($c \geq i$). Because we are primarily interested in compositionality, we assume that there are enough cache elements for each task, for the entire application ($C \geq N$), as well as for each optimization step ($c \geq i$). Moreover, at step i we do not have to compute the cache allocation for caches larger than *C-(N-i)*, because we know for sure that the rest of *N-i* tasks should also have at least one cache unit. For simplicity reasons we shortly denote the task T_i number of misses when having k allocated cache size, $miss(T_i, k)$, with m_i^k. We designate with $OPR_i^c = \{c_j\}_{(j=1,...,i)}$ the optimal cache partitioning ratio for the first i tasks of the application, having in total $\sum_{j=1}^{i} c_j = c$ cache. In this situation, the minimum number of misses is denoted with M_i^c. At the *i+1* iteration the solution OPR_{i+1}^c is build using the OPR_i^c solution from the previous step. Given a cache size c (*i+1* $\leq c \leq$ *C-(N-i-1)*), the task T_{i+1} has k cache units and the previous tasks $\{T_j\}_{(j=1,2,...,i)}$ have in total *c-k* cache units. Then the total number of misses of the *i+1* tasks is $M_{i+1}^c = M_i^{c-k} + m_{i+1}^k$. To determine the value of k that give the minimum value of M_{i+1}^c we perform an extensive search. The solution to $\mathcal{CAP}$ is obtained at step N and it is OPR_N^C.

In practice each task T_i might have a minimum cache limit under which T_i does not fulfill its deadlines anymore, denoted here with γ_i. As γ_i is dependent on the final product in which the application A is embedded, we consider that it is specified by the application designer. To express the minimum cache limit in the dynamic programming formulation, the following constraint applies:

$$c_i \geq \gamma_i. \tag{3.2}$$

Algorithm 1: Optimal cache partitioning ratio determination

foreach *i=1, 2, ..., N* **do**
  foreach *c=i, i+1, ..., C-(N-i)* **do**
    Find k for which $M_i^c = \min_{k\ i\ c}\{M_{i-1}^{c-k} + m_i^k\}$;
    $OPR_i^c = (OPR_{i-1}^{c-k}, k)$;
  end
end

For clarity reasons we present in the following a simple example of how Algorithm 1 builds the solution. In this example we consider the case of set based cache partitioning when the cache sizes are limited to a power of two. The number of tasks is *N=3* and the maximum allocable cache size is *C=8*, and no minimum cache limit. The number of misses as function of the cache

	1	2	4	8
T_1	16	13	7	3
T_2	20	10	2	1
T_3	12	8	4	3

Table 3.1: Example: Tasks number of misses function of cache size.

		1	2	3	4	5	6	7	8
Step 1	misses	$M_1^1 = 16$	$M_1^2 = 13$	$M_1^3 = 13$	$M_1^4 = 7$	$M_1^5 = 7$	$M_1^6 = 7$	n.a.	n.a.
T_1	*OPR*	(1)	(2)	(2)	(4)	(4)	(4)	n.a.	n.a.
Step 2	misses	n.a	$M_2^2 = 36$	$M_2^3 = 26$	$M_2^4 = 23$	$M_2^5 = 18$	$M_2^6 = 15$	$M_2^7 = 15$	n.a.
T_1, T_2	*OPR*	n.a.	(1, 1)	(1, 2)	(2, 2)	(1, 4)	(2, 4	(2, 4)	n.a.
Step 3	misses	n.a.	n.a.	$M_3^3 = 48$	$M_3^4 = 38$	$M_3^5 = 34$	$M_3^6 = 31$	$M_3^7 = 26$	$M_3^8 = 23$
T_1, T_2, T_3	*OPR*	n.a.	n.a.	(1, 1, 1)	(1,2,1)	(1,2,2)	(2,2,2)	(1,4,2)	(2, 4, 2)

Table 3.2: Example: dynamic programming steps.

size for each of the three task is presented in Table 3.1. Table 3.2 depicts the optimal misses M_i^c at each step of the Algorithm 1 and the corresponding partitioning ratio. At Step 1 of the algorithm the first task is considered. The cache size of task T_1 can not be larger than *6*, because we know that T_2 and T_3 should also have available at least one cache set each. In Step 2, the case when the cache size is 1 is not applicable because there is not enough cache for both tasks. Moreover, the cache size cannot be 8 because then T_3 would not have any cache available. Let us take a closer look on how to calculate a OPR_2^c value. For instance, for OPR_2^3 there are two options: T_2 has $k = 1$ cache sets (thus T_1 has 2 cache sets) or T_2 has $k = 2$ cache sets (thus T_1 has 1 cache set). However, the minimum number of misses for the (T_1, T_2) combination with *c=3* cache sets is achieved when *k=2*, therefore $OPR_2^3 =$ *(1,2)*. At Step 3 the same type of calculation applies and the final solution for a cache of *8* lines is $OPR_3^8 =$ *(2, 4, 2)*.

3.5 Compositionality investigation metric

In order to verify that a system is compositional, we introduce *the number of inter-task conflict misses* ($cmiss$) metric. In the remainder of this thesis, unless explicitly specified, we address the inter-task conflict misses simply as conflict misses. Let us consider the case of a task T_i. The number of conflict misses of a task $cmiss(T_i)$ is the total number of misses caused to T_i by other tasks T_j, with T_i and T_j belonging to the same application A. To exactly detect which

T_j cache miss is causing a flush of a data item that T_i needs in the future, all the data items of the application have to be individually tracked. These data items are identified by their address. The address space of a CAKE platform is huge (a processor can access up to few giga bytes) and the number of L2 accesses that we are dealing with are very large (e.g. up to 10^8 for decoding 10 frames of mpeg2 video). Thus in practice such tracing is not possible in a reasonable amount of time. However, there is an easy method to obtain an upper bound ($CM(T_i)$) of the inter-task conflict misses number $cmiss(T_i)$. $CM(T_i)$ represents the number of times a task T_j flushes some T_i data out of the cache. This represents an upper bound of the conflict misses number, as in some cases the flushed data might not be needed in the future, so no miss is actually encountered. The number of times a task T_j flushes some T_i data out of the cache requires just a simple counting, and no individual address tracing, therefore it is easily obtainable in practical situations.

We extend the number of conflict misses definition to application level as follows. The number of conflict misses *CM(A)* of an application *A* represents the sum of the conflict misses experienced by each task T_i of the application *A*. In order to have an idea about how much these conflicts impact the entire system, the values presented on the experimental results section are actually relative to the *A* number of misses:

$$CM(A) = \frac{\left(\sum_{T_i \in A} CM(T_i)\right)}{miss(A)}, \tag{3.3}$$

where $miss(A)$ is the total number of misses experienced by *A*.

3.6 Experimental results

The experiments presented in this section have two goals: (1) to check the compositionality of the system and (2) to illustrate the cache partitioning implications in the system performance. The workload consists of six applications composed out of various media tasks, running on a CAKE platform with four TriMedia cores, as introduced in Chapter 2. The partitioning ratio is determined with the method presented in Section 3.4, that minimizes the overall application number of misses. For each of the two goals, we compare the behavior of a conventional cache, a set-based partitioned cache, and an associativity based partitioned cache. In the case of the performance investigation,

we compare also with the case of an infinite cache. The comparison with the performance of an infinite cache is interesting because it gives an idea about the maximum improvement that one can theoretically achieve by tuning the L2 cache. In our case it was enough to approximate an infinite L2 with a cache of 4 Mega Bytes (MB) size and 16 ways associativity, as the experienced the number of misses of such a large cache is very low, no significant misses difference is observed between a cache of 2 MB and one of 4 MB and no substantial misses variation occurs for caches larger than 4 MB. Moreover we also investigate the performance implication of the memory alignment and allocation needed for cache partitioning, as explained in Section 3.3.

We experiment various L2 cache configurations representative for the state of the art L2 caches. These configurations consist of all the combinations among L2 sizes from 256 Kilo Bytes (KB) to 2 Mega Bytes (MB) and associativity of 4, 8, and 16 and an infinite L2 cache. We fixed the largest investigated associativity (16 ways), to be a step ahead of the state-of-the art L2 associativity (for instance the L2 cache of an Intel Extreme Quad-Core Processor Q6000 produced in 2007 is 8 way associative [42]). The reader should note that for the L2 caches having only 4 ways, a compositional associativity based partitioning is not possible, because an applications consists of 4 data processing tasks and an operating system task, thus there are not enough ways such that each task has at least an exclusive one.

In the remainder of this section we first present the system compositionality evaluation and then the performance investigation.

3.6.1 Compositionality

In order to investigate the compositionality of the system we use the number of conflict misses metric *CM(A)* as defined in Section 3.5. The *CM(A)* are reported for three L2 cases: conventional shared (*shared*), set-based partitioned (*set part*), and associativity based partitioned (*assoc part*), when possible. In Table 3.3 we present the relative *CM(A)* for each of the 6 applications. The values presented in this subsection are relative to the corresponding application total number of misses. The last row in Table 3.3 represents the average over all applications number of conflict misses. Figure 3.5 illustrates the average relative conflict misses *CM(A)* (the last row in Table 3.3), function of the cache size and associativity, for the case of the shared cache. The partitioned cache cases are not presented in Figure 3.5 beause their values are very small (under 1%), thus practically ”invisible” next to the shared case.

		256K/4	256K/8	256K/16	512K/4	512K/8	512K/16	1M/4	1M/8	1M/16	2M/4	2M/8	2M/16	4M/16
A_1	shared	79.8%	77.7%	76.0%	61.8%	59.5%	57.7%	48.0%	40.3%	38.0%	37.4%	34.2%	30.9%	31.1%
	set part	0.70%	0.62%	0.22%	0.81%	0.75%	0.46%	0.97%	0.14%	0.48%	0.97%	0.09%	0.09%	
	assoc part		0.04%	0.05%		0.02%	0.02%		0.03%	0.05%		0.04%	0.05%	
A_2	shared	84.0%	82.2%	83.6%	77.1%	73.6%	72.5%	69.9%	66.3%	65.3%	62.6%	61.2%	60.2%	56.4%
	set part	0.70%	0.6%	0.2%	0.8%	0.7%	0.4%	0.9%	0.1%	0.4%	0.9%	0.1%	0.2%	
	assoc part		0.04%	0.05%		0.02%	0.02%		0.03%	0.05%		0.04%	0.05%	
A_3	shared	79.7%	76.4%	75.4%	87.3%	84.9%	83.2%	84.2%	78.8%	77.3%	69.6%	65.7%	63.1%	58.5%
	set part	0.78%	0.62%	0.22%	0.81%	0.75%	0.46%	0.97%	0.14%	0.48%	0.97%	0.09%	0.09%	
	assoc part		0.03%	0.02%		0.02%	0.03%		0.01%	0.06%		0.05%	0.02%	
A_4	shared	71.9%	70.8%	70.3%	70.9%	67.1%	68.4%	64.5%	63.8%	63.4%	60.9%	60.1%	58.5%	55.2%
	set part	0.11%	0.12%	0.10%	0.10%	0.08%	0.10%	0.07%	0.10%	0.10%	0.09%	0.09%	0.09%	
	assoc part		0.13%	0.12%		0.13%	0.15%		0.10%	0.10%		0.09%	0.07%	
A_5	shared	81.0%	79.2%	82.1%	77.2%	73.6%	73.4%	77.4%	70.6%	69.9%	64.7%	63.0%	64.1%	59.0%
	set part	0.22%	0.33%	0.27%	0.20%	0.22%	0.20%	0.28%	0.11%	0.22%	0.30%	0.28%	0.30%	
	assoc part		0.12%	0.11%		0.08%	0.07%		0.09%	0.06%		0.07%	0.05%	
A_6	shared	72.6%	71.9%	73.2%	78.4%	75.7%	73.2%	66.8%	67.7%	65.1%	50.3%	46.4%	47.7%	43.3%
	set part	0.07%	0.07%	0.06%	0.07%	0.08%	0.06%	0.07%	0.08%	0.06%	0.05%	0.04%	0.04%	
	assoc part		0.06%	0.07%		0.05%	0.05%		0.03%	0.04%		0.04%	0.04%	
$Average$	shared	77.8%	75.9%	76.5%	75.1%	71.9%	71.1%	68.0%	64.0%	62.8%	57.1%	54.9%	53.8%	50.4%
	set part	0.37%	0.36%	0.18%	0.44%	0.35%	0.27%	0.54%	0.11%	0.25%	0.47%	0.16%	0.17%	
	assoc part		0.11%	0.12%		0.05%	0.19%		0.05%	0.30%		0.05%	0.04%	

Table 3.3: Relative inter-task conflict misses function of cache dimensions.

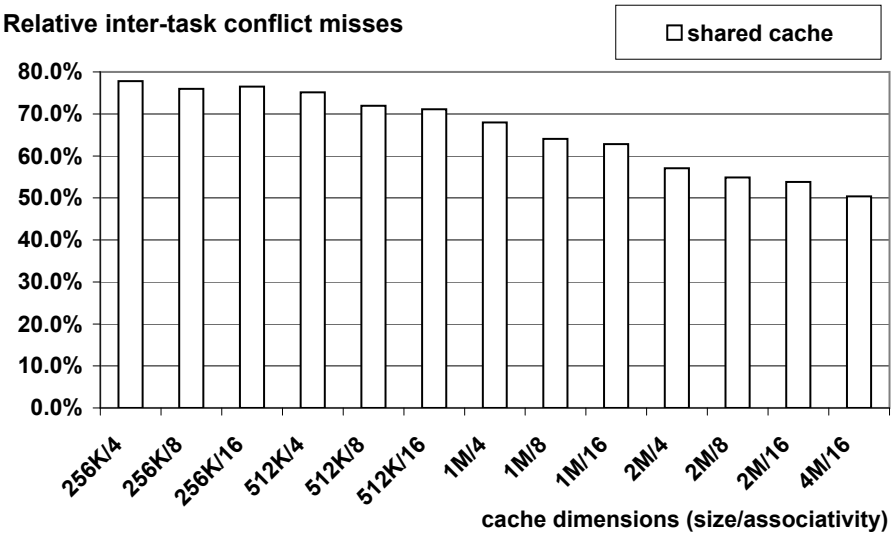

Figure 3.5: Average relative, inter-task conflict misses: shared cache

In Table 3.3 one can observe that both set based and associativity based partitioning techniques induce an extremely small inter-task conflict misses fraction (under 1% for each application in every cache combination). Nevertheless a small inter-task L2 interference can be observed even for the partitioned cache. This interference is due to the fact that the operating system has some shared data structures accessed by all tasks. Thus the OS's L2 partition is shared and exposed to the inter-task L2 interference. However these data structures are small (under 2KB) and typically fit in entirely in the OS's L2 partition, therefore the L2 interference is also small (more details about data sharing combined with cache partitioning are presented in the next chapter).

Unlike in a partitioned L2, in a shared L2 a large fraction of the misses represent actually inter-task conflict misses. We can see in Table 3.3 that the inter-task conflict misses range between a peak value of 87% (for A_3 with a 4 ways associtive L2 of 512KB) and a bottom value of 30% (for A_1 with 4MB of L2 cache). The average over all applications and all cache configuration is 66%. In general, the percentage of inter-task conflict misses decrease once the cache size is increased. This effect appears as a result of the fact that, the larger the cache, the more data fit in it and the total misses number decreases. One can also observe that the inter-task conflict misses number diminish when the L2 associativity is larger, for the same cache size. The explanation lies in the fact that, in general, the number of conflict misses is reduced due to the

availability of more ways, therefore the inter-task conflict misses follow the same trend. As visible in Figure 3.5 these trends are valid also on average over all the applications experimented. Nevertheless, in some cases (e.g. A_3), we notice an increase in inter-task conflict misses when increasing the cache (from 256KB to 512KB in the case of A_3). Moreover, A_5 deviates from the general trend because when having a 16 ways L2 of 256KB it experiences more inter-task conflict misses then with an 8 ways cache of the same size. The reasons for these deviations are not immediate, as the cache behavior is history based, and the shared cache is subject to unpredictable inter-task interference, However, what we can surely conclude from the compositionality investigation is that both set based and associativity based partitioning techniques can induce compositionality to the system, within 1% bounds.

3.6.2 Performance

As the experiments presented in the previous subsection suggest, both partitioning methods can induce compositionality. In our previous work [70] we present a brief quantitative comparison among the static set and associative cache partitioning, and here we extend this analysis. The purpose of the experiments presented in this subsection is to quantify the impact of the two types of cache partitioning on the system performance. As mentioned in Section 3.3, the prerequisites to cache partitioning are memory alignment and hierarchical dynamic memory allocation. Thus in this subsection we first investigate the impact on performance of those two memory related techniques. Second, we study the cache partitioning impact on performance. In the presented investigations we use two performance metrics, (1) the number of L2 misses per instruction (Misses Per Instruction, MPI), expressing the cache performance and (2) the average number of cycles required to execute an instruction (Cycles Per Instruction, CPI), expressing the total processing speed.

Memory allocation and alignment impact on performance

In order to gain inside in the impact on performance of the memory allocation and alignment (MAA) in general, for a conventional shared cache, we compare two cases: (1) the case with a common heap for all tasks, thus no special memory alignment and allocation implemented (*No MAA*) and (2) the case with the memory aligned and allocated according to the techniques described in Section 3.3 (*MAA*). For the last case we implement an hierarchical memory allocation strategy and we enforce a static memory alignment, as proposed.

		256K/4	256K/8	256K/16	512K/4	512K/8	512K/16	1M/4	1M/8	1M/16	2M/4	2M/8	2M/16	4M/16
	No MAA	0.077	0.059	0.037	0.025	0.020	0.018	0.012	0.012	0.012	0.009	0.009	0.009	0.000
A_1	*MAA*	0.048	0.039	0.037	0.020	0.017	0.014	0.012	0.011	0.010	0.009	0.009	0.009	0.000
	No MAA	0.132	0.120	0.126	0.064	0.058	0.057	0.032	0.030	0.029	0.024	0.022	0.018	0.000
A_2	*MAA*	0.164	0.135	0.132	0.071	0.059	0.058	0.032	0.030	0.029	0.022	0.021	0.020	0.000
	No MAA	0.095	0.090	0.096	0.029	0.027	0.026	0.011	0.010	0.009	0.006	0.005	0.005	0.000
A_3	*MAA*	0.118	0.100	0.099	0.040	0.028	0.027	0.015	0.011	0.010	0.006	0.005	0.005	0.000
	No MAA	0.310	0.241	0.236	0.148	0.090	0.084	0.038	0.038	0.042	0.030	0.027	0.025	0.000
A_4	*MAA*	0.153	0.182	0.184	0.090	0.085	0.082	0.051	0.049	0.049	0.037	0.036	0.034	0.000
	No MAA	0.126	0.100	0.101	0.078	0.047	0.047	0.027	0.026	0.024	0.011	0.010	0.007	0.000
A_5	*MAA*	0.117	0.102	0.111	0.062	0.049	0.048	0.035	0.026	0.025	0.010	0.009	0.007	0.000
	No MAA	0.083	0.083	0.086	0.027	0.023	0.023	0.013	0.012	0.012	0.016	0.012	0.015	0.000
A_6	*MAA*	0.088	0.084	0.089	0.027	0.024	0.022	0.013	0.014	0.013	0.016	0.012	0.014	0.000
	No MAA	0.137	0.116	0.114	0.062	0.044	0.043	0.022	0.021	0.021	0.016	0.014	0.013	0.000
$Average$	*MAA*	0.115	0.107	0.109	0.052	0.044	0.042	0.026	0.023	0.023	0.016	0.015	0.014	0.000

Table 3.4: 100xMPI: no special memory alignment & allocation vs. proposed memory alignment & allocation

		256K/4	256K/8	256K/16	512K/4	512K/8	512K/16	1M/4	1M/8	1M/16	2M/4	2M/8	2M/16	4M/16
	No MAA	1.52	1.47	1.40	1.32	1.30	1.30	1.27	1.27	1.27	1.27	1.27	1.27	1.27
A_1	*MAA*	1.42	1.40	1.40	1.32	1.30	1.30	1.27	1.27	1.27	1.27	1.27	1.27	1.27
	No MAA	1.80	1.70	1.75	1.35	1.35	1.32	1.22	1.22	1.22	1.17	1.12	1.12	1.12
A_2	*MAA*	1.95	1.77	1.82	1.4	1.35	1.32	1.22	1.22	1.22	1.12	1.12	1.12	1.12
	No MAA	1.52	1.5	1.57	1.2	1.17	1.17	1.1	1.1	1.1	1.07	1.07	1.07	1.07
A_3	*MAA*	1.67	1.57	1.55	1.2	1.17	1.20	1.12	1.10	1.10	1.07	1.07	1.07	1.07
	No MAA	3.67	2.85	2.80	2.00	1.65	1.65	1.40	1.37	1.40	1.32	1.32	1.32	1.32
A_4	*MAA*	2.42	2.40	2.30	1.65	1.65	1.62	1.45	1.45	1.32	1.32	1.32	1.32	1.32
	No MAA	1.75	1.65	1.63	1.42	1.40	1.27	1.17	1.17	1.12	1.12	1.12	1.10	1.10
A_5	*MAA*	1.70	1.70	1.55	1.32	1.27	1.27	1.22	1.17	1.15	1.20	1.12	1.10	1.10
	No MAA	1.50	1.47	1.47	1.20	1.17	1.17	1.10	1.10	1.10	1.10	1.10	1.10	1.10
A_6	*MAA*	1.55	1.47	1.47	1.17	1.17	1.17	1.10	1.10	1.10	1.10	1.10	1.10	1.10
	No MAA	1.96	1.77	1.78	1.42	1.34	1.31	1.21	1.21	1.20	1.18	1.17	1.16	1.16
$Average$	*MAA*	1.79	1.69	1.64	1.34	1.32	1.32	1.23	1.22	1.19	1.18	1.17	1.16	1.16

Table 3.5: CPI: no special memory alignment & allocation vs. proposed memory alignment & allocation.

These mechanisms ensure that the cache misses experienced by a task do not depend on other task's memory allocation history. The complete results are presented in Table 3.4 and 3.5 for the L2 MPI and CPI, respectively. The average MPI and CPI over the six applications are presented in Figure 3.6 and 3.7, respectively.

By applying a special MAA, the addresses of tasks' variables change. Thus tasks access the cache following a different pattern than in the case no special MAA is employed. When looking to multitasking application, this access pattern change has implications in (1) the misses intrinsic to each task and (2) the inter-task cache interference. The performance difference among the *MAA* and the *No MAA* case is given by the summation of these two effects and it is application dependent.

When looking at cache sizes one can observe that, on average, for small caches the utilization of memory alignment and allocation is beneficial for performance. For an L2 of 256KB the average MPI reduction (over all presented associativities) is 0.00012, representing 10% from the average *No MAA* MPI. This causes a CPI decrease of 0.11 representing 6% from the *No MAA*'s CPI, averaged over all associativities. Over all targeted L2 sizes and associativities, the maximum performance improvement caused by MAA is encountered in the case of 265KB, 4 ways associative L2. For this L2 configuration the applications experience a reduction of 0.000023 MPI (20% from the corresponding *No MAA*'s MPI) and 0.18 CPI (10% from the *No MAA*'s CPI). For a 512KB L2 the average MPI reduction is 0.00003, representing 7% from the *No MAA* MPI. This cause an CPI decrease of 0.03 representing 2% from the *No MAA*' CPI. For an 1MB L2 the MAA induces a slight performance loss. On average over all investigated associativities the *No MAA* case exhibits a 0.00004 increase in MPI when compared with the corresponding *MAA* case. This decrease represents 7% from the average *No MAA* MPI. This causes an average CPI decrease of 0.03, representing 2% from the *No MAA*' CPI. For caches larger than 1MB the difference between the two cases is under 0.01 CPI because most of the application's data fit in the cache regardless the alignment.

When looking at associativity we can notice that the MAA impacts more the low associativity caches. For a 4 ways associative L2 on average (over all investigated size) the MPI reduces with 0.0001 when applying MAA, causing a 0.8 CPI decrease. Relative to the corresponding average *No MAA* values, these reductions translate in 13% and 5% for MPI and CPI, respectively. For a 8 ways associative L2 on average the performance of the MAA scheme is still better than the one of without MAA, however the differences are smaller

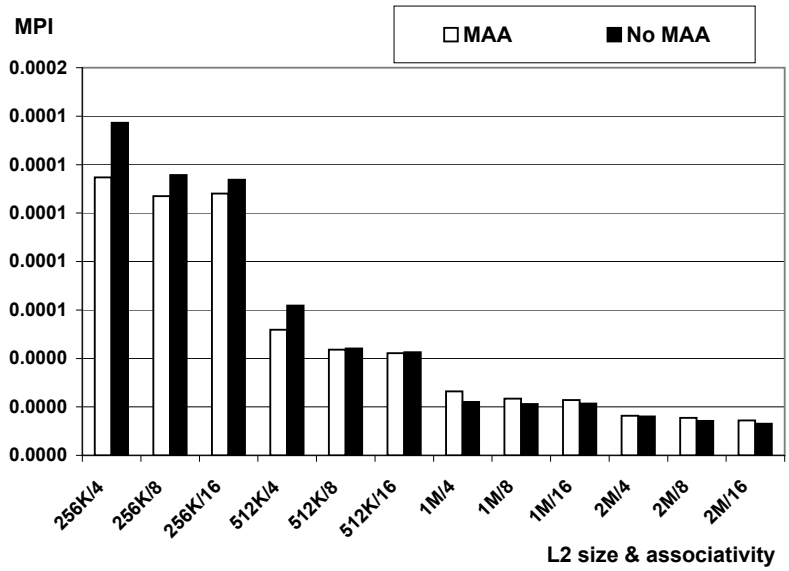

Figure 3.6: Average MPI: no special memory alignment & allocation vs. proposed memory alignment & allocation.

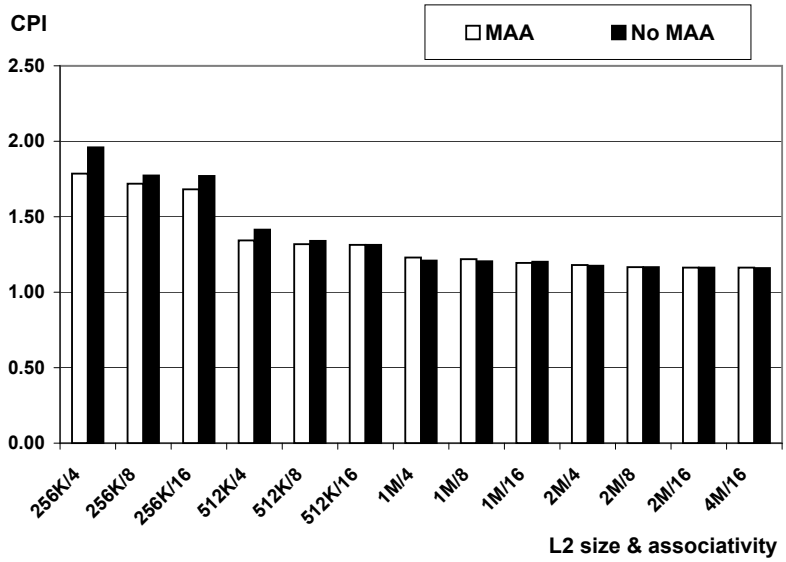

Figure 3.7: Average CPI: no special memory alignment & allocation vs. proposed memory alignment & allocation.

than the ones encountered for a 4 ways associative L2. On average the MPI reduces with 0.00002 when applying MAA, causing a 0.2 CPI decrease. Relative to the corresponding average *No MAA* values, these reductions translate in 4% and 1% for MPI and CPI, respectively. For a 16 ways associative L2 the performance implications of the MAA scheme are negligible (under 0.01 CPI).

When looking at each application, one can see in Tables 3.4 and 3.5 that A_1 and A_4 are always positively impacted by the MAA, regardless the L2 dimensions. On average over all L2 sizes and associativities, the A_1's MAA MPI represents 21% from the MPI of the A_1's without MAA. In terms of CPI, this corresponds with a 2% reduction. In the case of A_4 the relative average MPI and CPI reduce with 24% and 14%, respectively. The applications A_2, A_3, A_5, and A_6 are marginally influenced by the MAA scheme. Their summed MPI and CPI increases slightly, with 5% and 1%, respectively, relative to the MPI/CPI sum for the *No MAA* case.

An important observation is that for small caches and for reduced associativity caches the impact (either positive or negative) of MAA is larger than for large size and/or associativity caches. The reason behind this is that in small and/or low associativity caches a large fraction of the misses are conflict misses. Here the term "conflict miss" is used in the sense defined in [38], and such a miss is not necessarily an inter-task conflict miss, but it is in general a miss occurring due to multiple addresses mapping in the same cache line. The application's accessing pattern has a great influence on the number of conflict misses. Thus by employing an MAA scheme, the accessing pattern is largely impacted therefore so is also the number of conflict misses.

Cache partitioning impact on performance

In this subsection we determine which type of cache partitioning performs better for the media domain.

The results corresponding to each of the 6 applications are detailed in Table 3.6 and Table 3.7, for MPI and the CPI, respectively. Figures 3.8 and 3.9 illustrate the average over the 6 applications for the MPI and CPI, respectively. We compare the performance of a shared cache (*shared*) with the one of a set based partitioned cache (*set part*), and the one of an associativity based partitioned cache (*assoc part*). In all the above mentioned cases the used memory allocation and alignment are the ones presented in Section 3.3.

Two phenomena are behind the difference in misses number between a

shared and a partitioned cache. If the cache is partitioned, the inter-task conflict misses are eliminated (which means in total less misses) but every task can use less cache space than in the shared case (which means more misses). As one can notice in Tables 3.6 and 3.7, and Figures 3.8 and 3.9, the set based partitioned cache performs usually better than both the shared and associativity based partitioned L2. The performance difference is larger for small L2 situations (256KB to 1MB) and tends to flatten for large caches (from 2MB to the infinite cache). This is an expected behavior, as larger caches may encompass more of the applications data, which causes fewer misses for all cache configurations, therefore the performance differences among them are smaller.

When compared with a conventional L2, we found that the set based partitioned L2 usually performs better, achieving up to 62% MPI reduction corresponding to an up to 31% CPI decrease (relative to the corresponding shared cache MPI and CPI values). In absolute values, these maximum improvements are 0.0008 for MPI and 0.75 for CPI.

On average over all the applications, the set based partitioned cache has less misses than the shared cache. When L2 is larger or equal to 2 MB the difference in CPI among the partitioned and the shared cache is less than 1%. For caches smaller than 2 MB, the set based partitioning brings the following MPI reductions, absolute and relative to the corresponding shared MPI values: (1) 0.00045 and 41%, respectively, for an L2 of 256 KB, (2) 0.00009 and 17%, respectively, for an L2 of 512 KB, and (3) 0.00008 and 29%, respectively, for an 1 MB L2. All these values are an average over all the inquired L2 associativities. In terms of CPI this translates into the following average decreases (again in absolute and relative values): (1) 0.32 and 18%, respectively, for an L2 of 256 KB, (2) 0.06 and 4%, respectively, for an L2 of 512 KB, and (3) 0.03 and 2%, respectively, for an 1 MB L2. On average for caches smaller than 2MB, when the L2 is set-based partitioned it experiences 29% less misses per instructions and 8% less cycles per instructions.

Moreover, some applications are more sensitive to cache partitioning than others. From all the 72 studied combinations of applications, L2 sizes and all associativities, only in 8 cases the MPI and the CPI of the set based partitioned L2 is larger than the one of the shared L2. The maximum MPI increase when using a partitioned L2 compared with the shared L2 is 0.00011. This represents 13% from the maximum possible MPI reduction presented in the previous paragraph. The corresponding maximum CPI increase is 0.03, representing 4% maximum possible CPI decrease induced by set based partitioning. On average over the 8 cases, these performance reductions are 19% and 4% for

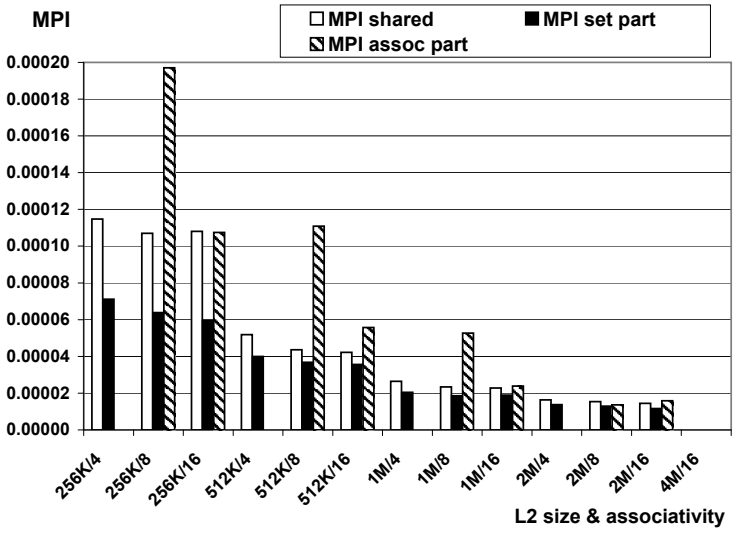

Figure 3.8: Average MPI: shared vs. set partitioning vs. associativity partitioning

MPI and CPI, respectively. These numbers suggest that the number of cases when set-based brings a significant performance benefits are way more frequent that the cases when it brings performance degradation. Moreover, the encountered degradations can be considered not significant.

Furthermore, we assess the fraction from the maximum possible performance boost that is brought by splitting the cache, to give an idea about the relative performance impact of partitioning. For this we define the maximum CPI improvement achievable by tuning the L2 cache as the difference between the CPI obtained with a conventional shared L2 and the one obtained with an infinite L2: $\Delta CPI_{max} = CPI_{shared} - CPI_{infinite}$. The CPI improvement caused by set based partitioning is the difference among the CPI in the shared L2 case an the one in the partitioned case: $\Delta CPI_{set\,part} = CPI_{shared} - CPI_{set\,part}$. Figure 3.10 presents the percentage of CPI improvement due to set based partitioning from the maximum achievable ($\frac{\Delta CPI_{set\,part}}{\Delta CPI_{max}}$), depending on the cache size. We do not present the results for caches larger than 1 MB, as for those the CPIs are the same for a shared, partitioned, and infinite caches. When averaging the percentages from Figure 3.10 we found that set based partitioning brings on average 48% of the possible CPI improvement, while keeping the same cache size.

		256K/4	256K/8	256K/16	512K/4	512K/8	512K/16	1M/4	1M/8	1M/16	2M/4	2M/8	2M/16	4M/16
A_1	shared	0.048	0.039	0.037	0.020	0.017	0.014	0.012	0.011	0.010	0.009	0.009	0.009	0.000
	set part	0.030	0.027	0.025	0.014	0.010	0.009	0.009	0.009	0.009	0.008	0.008	0.008	
	assoc part		0.050	0.038		0.032	0.013		0.011	0.011		0.010	0.010	
A_2	shared	0.164	0.135	0.132	0.071	0.059	0.058	0.032	0.030	0.029	0.022	0.021	0.020	0.000
	set part	0.091	0.060	0.053	0.037	0.038	0.037	0.026	0.027	0.027	0.019	0.016	0.016	
	assoc part		0.204	0.137		0.123	0.073		0.066	0.029		0.026	0.021	
A_3	shared	0.118	0.100	0.099	0.040	0.028	0.027	0.015	0.011	0.010	0.006	0.005	0.005	0.000
	set part	0.061	0.069	0.065	0.036	0.035	0.034	0.011	0.008	0.008	0.005	0.003	0.002	
	assoc part		0.155	0.076		0.100	0.056		0.074	0.010		0.011	0.005	
A_4	shared	0.153	0.182	0.184	0.090	0.085	0.082	0.051	0.049	0.049	0.037	0.036	0.034	0.000
	set part	0.110	0.108	0.107	0.073	0.071	0.069	0.036	0.033	0.035	0.026	0.025	0.024	
	assoc part		0.296	0.123		0.152	0.101		0.100	0.042		0.000	0.028	
A_5	shared	0.117	0.102	0.111	0.062	0.049	0.048	0.035	0.026	0.025	0.010	0.009	0.007	0.000
	set part	0.062	0.046	0.041	0.043	0.032	0.031	0.026	0.024	0.022	0.013	0.012	0.007	
	assoc part		0.237	0.137		0.140	0.032		0.039	0.037		0.019	0.014	
A_6	shared	0.088	0.084	0.089	0.027	0.024	0.022	0.013	0.014	0.013	0.016	0.012	0.014	0.000
	set part	0.071	0.073	0.066	0.037	0.035	0.034	0.015	0.012	0.012	0.012	0.012	0.012	
	assoc part		0.239	0.134		0.118	0.059		0.025	0.014		0.015	0.017	
$Average$	shared	0.115	0.107	0.109	0.052	0.044	0.042	0.026	0.023	0.023	0.017	0.015	0.015	0.000
	set part	0.071	0.064	0.060	0.040	0.037	0.036	0.020	0.019	0.019	0.014	0.013	0.012	
	assoc part		0.197	0.107		0.111	0.056		0.053	0.024		0.014	0.016	

Table 3.6: 100xMPI: shared vs. set partitioning vs. associativity partitioning (6 applications).

		256K/4	256K/8	256K/16	512K/4	512K/8	512K/16	1M/4	1M/8	1M/16	2M/4	2M/8	2M/16	4M/16
A_1	shared	1.42	1.40	1.40	1.32	1.30	1.30	1.27	1.27	1.27	1.27	1.27	1.27	1.27
	set part	1.37	1.32	1.32	1.27	1.27	1.27	1.27	1.27	1.27	1.27	1.27	1.27	
	assoc part		1.45	1.35		1.37	1.27		1.27	1.27		1.27	1.27	
A_2	shared	1.95	1.77	1.82	1.40	1.35	1.32	1.22	1.22	1.22	1.12	1.12	1.12	1.12
	set part	1.42	1.27	1.35	1.20	1.22	1.20	1.15	1.17	1.17	1.12	1.12	1.12	
	assoc part		2.40	1.85		1.65	1.37		1.37	1.17		1.12	1.12	
A_3	shared	1.67	1.57	1.55	1.20	1.17	1.20	1.12	1.10	1.10	1.07	1.07	1.07	1.07
	set part	1.30	1.30	1.30	1.17	1.17	1.17	1.07	1.07	1.07	1.07	1.07	1.07	
	assoc part		2.25	1.37		1.55	1.25		1.37	1.1		1.07	1.07	
A_4	shared	2.42	2.40	2.30	1.65	1.65	1.62	1.45	1.45	1.32	1.32	1.32	1.32	1.32
	set part	1.67	1.65	1.65	1.57	1.47	1.45	1.37	1.32	1.32	1.32	1.32	1.32	
	assoc part		6.35	1.8		2.4	1.7		1.82	1.37		1.32	1.32	
A_5	shared	1.70	1.70	1.55	1.32	1.27	1.27	1.22	1.17	1.15	1.20	1.12	1.10	1.10
	set part	1.35	1.30	1.25	1.27	1.27	1.22	1.22	1.20	1.17	1.20	1.12	1.10	
	assoc part		3.30	1.77		2.15	1.22		1.27	1.25		1.20	1.12	
A_6	shared	1.55	1.47	1.47	1.17	1.17	1.17	1.10	1.10	1.10	1.10	1.10	1.10	1.10
	set part	1.40	1.37	1.30	1.20	1.20	1.20	1.10	1.10	1.10	1.10	1.10	1.10	
	assoc part		3.92	1.47		1.77	1.30		1.20	1.12		1.10	1.10	
$Average$	shared	1.79	1.69	1.64	1.34	1.32	1.32	1.23	1.22	1.19	1.18	1.17	1.16	1.16
	set part	1.42	1.37	1.36	1.28	1.27	1.25	1.20	1.19	1.18	1.18	1.17	1.16	
	assoc part		3.28	1.60		1.82	1.35		1.38	1.21		1.18	1.17	

Table 3.7: CPI: shared vs. set partitioning vs. associativity partitioning (6 applications).

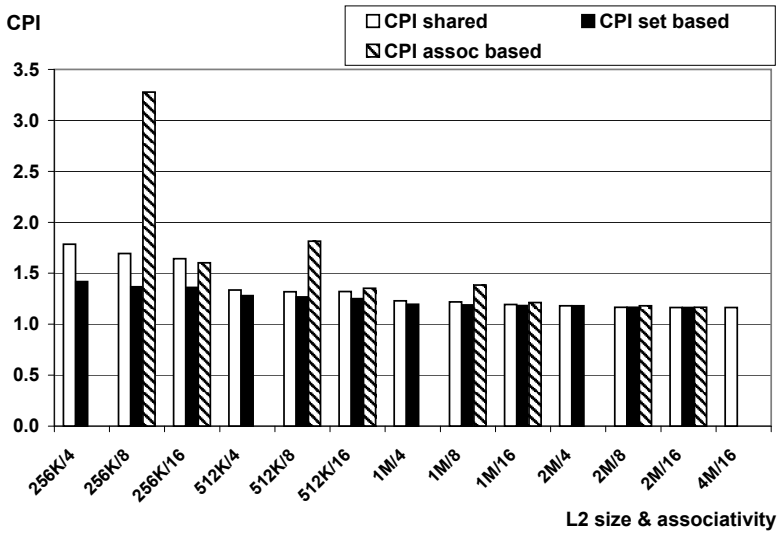

Figure 3.9: Average CPI: shared vs. set partitioning vs. associativity partitioning

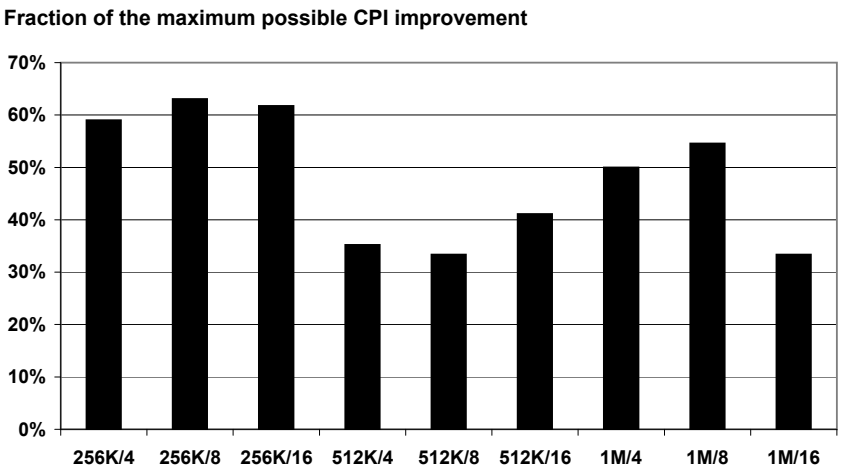

Figure 3.10: Set based partitioning CPI improvement: fraction from the maximum possible

In the case of associativity based partitioning, we found that the partitioned cache performs in general worse than a conventional shared cache. Concretely, associativity based partitioning leads to up to 187% MPI increase corresponding to up to 229% CPI degradation, when compared to a shared cache of the same size. In absolute values, this maximum performance loss are 0.00135 MPI and 3.95 CPI.

As it also is the case for set-based partitioning, here the performance differences tend to be smaller when the cache is larger as well. For a cache larger or equal to 2 MB the performance difference among an associativity partitioned cache and a shared cache is under few percent. For a cache smaller than 2 MB, on average over all applications, the associativity based partitioning leads to 0.00033 more misses per instruction than the shared cache, which results in a 0.37 larger CPI. Relative to the average shared cache MPI and CPI, these values represent 52% and 25%, respectively. Compared with a set based partitioned cache, on average the associativity based one experiences 0.0005 more misses per instruction and 0.49 larger CPI. Relative to the numbers for the associativity based partitioned cache, these increases represent 55% and 27% for the MPI and CPI, respectively.

As expected, for a fixed cache size, the performance of the associativity based partitioned cache increases when the associativity of the cache is larger. The reason for this tendency is that in a cache with larger associativity the granularity of the partitioning can be finer, therefore the partitioning ratio can be better tuned to the cache requirements of each task. Only in few cases (five) when the cache is 16 way associative (the largest investigated associativity), the partitioned cache performs better than the shared cache. At most these improvements reach a value of 0.00061 for the MPI (33% from the MPI of the shared L2) and 0.5 for the CPI (21% from the CPI of the the shared L2), respectively. On average, these improvements represent 18% and 7% from the MPI, respectively, the CPI of the corresponding average shared L2. These results indicate that, in general for associativity based partitioning the compositionality comes with a large performance price.

These results for associativity based partitioning might seem surprising when compared with previous work using the same type of cache partitioning [43],[20], [103]. All these articles report performance improvements caused by this type of cache partitioning. The reason for this opposite performance trend observed in our experiments relates to the fact that in the mentioned methods cache line sharing is allowed in a certain degree, in order to utilize the cache space at maximum. However, cache sharing (in whatever

degree) is exactly what we want to avoid because it perturbs the system compositionality. In our case each and every task has to have its own exclusive cache ways. We remind that in the exclusive partitioned situation every task can use less cache space than when some lines are shared, which might result in more misses. The fundamental difference with previous work in this field is that they target performance whereas we target compositionality.

Overall, for the considered applications we found that the set based cache partitioning is always performing better than the associativity based cache partitioning. This is explained by the fact that associativity based cache partitioning is decreasing the number of ways a task can use. It is known that, having a fixed cache size, a cache organization with a larger associativity (and a small number of sets) performs most of the times better than one with less associativity (but more sets) [37], [38]. For a clear understanding of this phenomenon, let us look at the two extremes of cache organization, namely the fully associative cache and the direct mapped cache. The fully associative cache has the potential to perform at least as good as a direct mapped cache of the same size. When data have to be flushed from the cache in a fully associative organization the cache controller can replace any line, usually one that is not likely to be needed in the near future. In a direct mapped cache there is no freedom to pick the replaced line, so data that were accessed in the past and it is likely to be required in the near future might be swapped out, causing more misses. This effect can be also observed in Tables 3.6 and 3.7, where for a given cache size, when the associativity increases the performance increases. In a set based partitioning scheme the number of cache sets that each task may use is reduced, but the associativity is kept the same, so this scheme is likely to have a better performance than the associativity-based one.

3.7 Conclusion

In this chapter we introduced the idea of task centric memory management for an embedded multiprocessor and we presented the options for enforcing such a management scheme. In our scheme we use caches (hardware controlled memories) for flexibility reasons. To ensure performance compositionality the cache is exclusively partitioned among the application tasks. Starting from the conventional cache organization, we identified two options for cache partitioning, namely set based and associativity based partitioning. For the set based partitioning we proposed a new implementation method tailored to a shared L2, consisting of the following: (1) a hardware address translation mecha-

nism that redirects the accesses of each task in a part of the cache exclusively assigned to it, (2) a set of cache reservation software application programmer interfaces that can be used at initialization time to allocated cache for each task (3) a memory alignment and allocation scheme to ensure that, even if tasks dynamically allocate memory in an interleaved fashion, without a fixed allocation order (i.e. a task may get different addresses for its variables, from a run to the another, depending on other tasks speeds and the moments when they perform memory allocation), the cache remains compositional. Subsequently to cache partitioning, we proposed a technique to find the cache partitioning ratio that minimizes the overall number of misses (based on a Dynamic Programming formulation).

To practically investigate the capabilities of these partitioning methods we used a CAKE platform with L2 sizes ranging from 256 Bytes to 2MBytes. For this investigation we utilized a benchmark consisting of six application, each of which composed by four media tasks. We measured and compared the application compositionality and performance in three cases: a shared conventional cache, a set based partitioning, and an associativity based partitioning. Moreover, we also discussed the impact of the memory alignment and allocation required for partitioning, on the L2 performance. To quantify the compositionality we utilized the number of inter-task conflict misses metric. As performance metrics we utilized the number of misses per instruction and the processors' average cycles per instruction.

The experiments suggest that both cache partitioning methods are potential candidates to separate the tasks in cache and induce compositionality, as their conflict misses represent less than 1% from the total application misses. However, the set based method offers, for the same cache dimensions, more allocable cache units than the associativity based one, as in a state-of-the-art cache the number of sets is few orders of magnitude larger than the number of ways. This is a property of major importance, as the number of tasks comprised by a system has a clear tendency to increase. As expected, we observed that the conventional cache's compositionality is very poor (i.e., on average 66% of an application misses are actually conflict misses).

With respect to performance we found that, when compared to a shared L2, the set based cache partitioning improves the performance of the systems in 89% of the cases studied (64 out of 72). In this cases we found that, compared with a shared L2, the set-based partitioned one provides up to 62% less L2 misses per instruction and 31% faster computation, with an average of 29% and 8%, respectively. Moreover, in the few (8 out of 72) cases when the set-based

partitioning degrades the L2 performance the average of these degradations is not large, 19% for the MPI and 4% for CPI.

Moreover we found that associativity based partitioning degrades the memory hierarchy performance, in most of the cache configurations. When compared with the conventional cache organization, associativity based partitioning increases the number of misses per instruction with up to 187% slowing down the computation with 229%, with an average penalty of 52% more MPI and 25% less CPI.

Furthermore, the experiments indicate that when the required memory alignment and allocation is applied to a shared L2, its performance improves, on average. For an L2 under 1MB the average relative MPI reduction is 8.5%, leading to a 4% smaller CPI. For an L2 of 1MB the memory alignment and allocation induces a small performance loss of 7% in MPI and 2% in CPI. For caches larger than 1MB the difference between the two cases in negligible (under 0.01 CPI). In general we observed that for a small size or small associativity of the L2, the impact (either positive or negative) of the memory alignment and allocation is larger than for a large size and/or associativity L2.

As a result of this study, we conclude that the superiority of set based cache partitioning over associativity based partitioning is two fold: the available number of allocable cache parts is larger, and it does not reduce the associativity of a task's cache part. Subsequently, the performance of a set based partitioned L2 is better than the one of an associativity based partitioned L2. Therefore, in the next chapter we utilize set based partitioning at the base of the envisaged task centric cache management strategy.

Chapter 4

Task centric cache management

In previous chapter we introduced the idea of task centric cache management for the purpose of achieving compositionality. We identified two types of cache partitioning (namely associativity and set based partitioning), that can be utilized as a foundation for task centric management. These methods are directly applicable when tasks do not shared data and/or instructions. In media application, however, sharing of data and/or instructions is often present (for example: frame buffers in video applications, communication buffers in streaming applications, etc.). When tasks have common regions, i.e., shared data and/or instructions, ensuring compositionality poses extra challenges. Tasks ”decoupling” via cache partitioning is difficult, because their interdependence is inherent when some data or instructions are common to multiple tasks.

At the first glance, we identify two straightforward possibilities to support common regions when having task centric cache management. The first possibility is that a common region resides in a shared cache part. The shortcomings of this possibility is a loss of compositionality because when multiple tasks access this cache part, they can unpredictably flush each others data. Alternatively, the second possibility is that a common region resides in the cache part of each task using it. This certainly provides compositionality, however, coherence problems and poor cache utilization occur because of data replication in cache. For example, when the instructions of the H.264 parallel tasks are multiplied in cache, our experiments indicate a 50% number of misses increase, resulting in 10% execution time penalty, when compared with the case when only one copy of that code is cached (a conventional shared cache). Because both these possibilities have unacceptable shortcomings in terms of

compositionality or performance, there is a need for a new cache management method that effectively supports sharing.

In this chapter we propose a tasks centric cache management method that supports sharing, and achieves compositionality within reasonable bounds [71], [72]. Our method mainly uses set-based partitioning, as recommended by the results in the previous chapter. However, inside the shared cache regions another level of inter-task separation is imposed (by means of associativity based cache partitioning) to limit the intrinsic inter-task conflicts. We use a novel, set and associativity based partitioning scheme (denoted in the rest of this thesis as "mixed" partitioning).

Subsequent to the compositionality issue is the performance optimization issue. After the cache is split among tasks, the natural question that arises is how much cache each task should have. In media applications both execution time and throughput are of major importance. In this line of reasoning we propose two methods for tuning the cache partitioning ratio accordingly to these two criteria. First, we extend the method to minimize the number of misses presented in Chapter 3 such that it supports common regions. Second, we introduce an optimization method based on simulated annealing to find the cache partitioning ratio that maximizes the throughput [69].

The outline of this chapter is as follows. In Section 4.1 we introduce a mixed set and associativity based cache partitioning targeted for tasks that have common regions, followed by a detailed description of the compositional caching for these regions in Section 4.2. In Section 4.3 we discuss the implementation of the mixed cache partitioning technique. Then in Section 4.4, we tackle the problem of finding the best cache partitioning ratio. We first present an extension of the algorithm for optimizing the number of misses (already described in Chapter 3) such that it supports also sharing and then we introduce a new algorithm for maximizing the throughput. Section 4.5 describes the experimental results and finally in Section 4.6 we draw the conclusions of this chapter.

4.1 Mixed cache partitioning

This section details the primal part of the task centric management scheme: the mixed cache partitioning. The first step toward achieving an appropriate cache partition is to ensure that the instances of private tasks data and common regions don't trash each other in cache. For that we restrict the cache parts used by every task and by every common region. We chose the set based partitioning

to isolate different tasks and common regions footprints in the cache, because this partitioning type proved (1) to be the best performance-wise and (2) to provide enough means for its utilization in real life multimedia applications (see Chapter 3).

After the separation of tasks and common regions in cache, we create the premises such that tasks don't trash each other data/instructions in the cache sets of the common regions. To avoid trashing in the shared cache sets we present two possible approaches:

- The cache allocated to the common region is as large as the region itself. In this case no misses occur, hence no unpredictable trashing can be present.

- Inside the cache sets of a common region, tasks may use the data/instructions if they are already there (share) but on a miss they are not allowed to flush other tasks data/instructions (don't interfere).

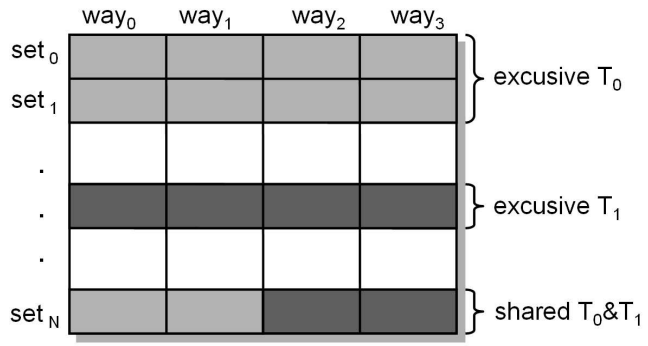

Figure 4.1: Mixed cache partitioning

The first solution depends on the application and on the available cache, so it may be not always applicable. For instance, state of the art video reference frame buffers typically do not fit in the cache. The second solution is more general and can be applied regardless of the relation between the sizes of the available cache and the size of the common data. This general solution can be implemented by allocating to each task sharing a region a number of ways in the sets allocated to that common region.

Summarizing, to achieve compositionality we use mixed cache partitioning as illustrated for a simple example in Figure 4.1. In the shared T_1 and T_2 cache region tasks can query all the four ways of the corresponding cache set for a possible hit. However, if for example a T_2 access misses in cache, the data

replacement may take place only in the ways which are T_2's private, i.e., way_2 and way_3.

In the next section we detail on the common region caching and we explain the relation between mixed cache partitioning and cache locality.

4.2 Compositional sharing of data/instructions in cache

This subsection gives inside in the cache mechanism that we devise to support inter-task data/instructions sharing. For this purpose, we first introduce a simple example, to illustrate the common region cache isolation induced by associativity based partitioning. Subsequently, we show that cache partitioning enables a full exploitation of spatial locality and potentially increases the benefits brought by prefetching techniques. Finally, we present the requirements of our method in terms of cache organization.

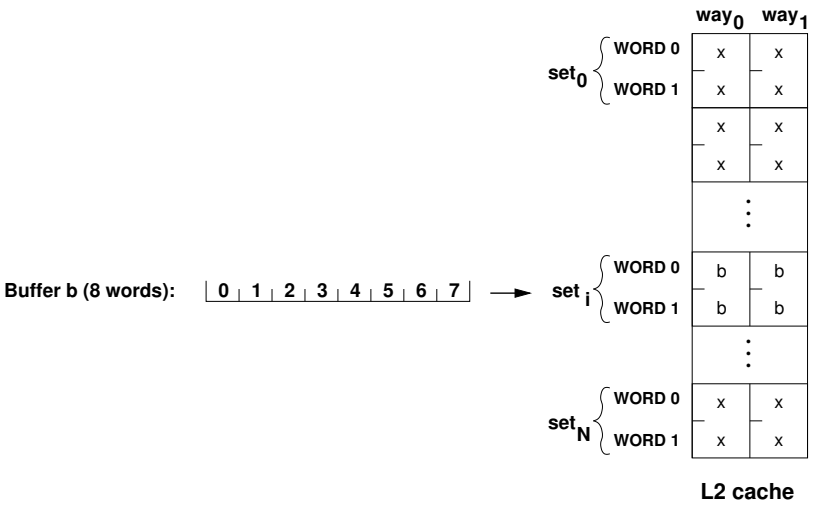

Figure 4.2: Example: buffer "b" and its allocated cache set

Let us assume that the data in the buffer b (Figure 4.2) are produced by a task T_P and consumed by another task T_C, in a "First In First Out" (FIFO) manner. We furthermore assume that b has allocated one set of cache (set_i in Figure 4.2) and the cache is two ways associative. Moreover, in this example we consider that a cache line can contain two consecutive elements of b. As no

conventional cache line replacement policy is aware of the application's task structure, the inter-task cache flushing problem is independent of the replacement policy utilized. For the sake of the example, we assume a "last recently used" replacement policy.

The sequences (A), (B), (C), and (D) in Figures 4.3 and 4.4 present the cache activity inside the set associated to the buffer *b*, as the result of the accesses of tasks T_P and T_C. Figure 4.3 corresponds to the case when the cache ways of *b* are shared among its producer and its consumer, and Figure 4.4 corresponds to the case when the cache is partitioned, therefore T_P and T_C use just one cache way each. In all the (A), (B), (C), or (D) cache images, the elements of buffer *b* that miss in the cache at the last access are encircled and the ones already produced/consumed are depicted in bold. Let us assume that initially the producer T_P produces *b[0]* to *b[4]*. This results in three misses corresponding to *b[0]*, *b[2]*, and *b[4]* (*b[1]*, *b[3]*, and *b[5]* being loaded in the cache as result of spatial locality). After these T_P accesses the cache contains the elements *b[2]* to *b[5]*, as visible in Figure 4.3(A). Then the consumer T_C accesses *b[0]* to *b[3]* and will flush all the elements of *b* cached for T_P. The *b[5]* data item was not yet produced, so when T_P continues its execution it has a miss and *b[4]* and *b[5]* are reloaded in cache. However, if T_P would have produced the element *b[5]* before the consumption of *b[2]* and *b[3]*, there would have been no cache miss for *b[5]*. As one can see, in the case of a shared cache, the number of cache misses is dependent on the tasks speeds and task scheduling. This number of misses cannot be predicted without a very detailed knowledge about task scheduling and tasks' execution times. Such knowledge is difficult to acquire especially in a multi-processor systems with natural load balancing, like the one we consider, thus in the system we consider the number of misses in the shared buffer is practically unpredictable.

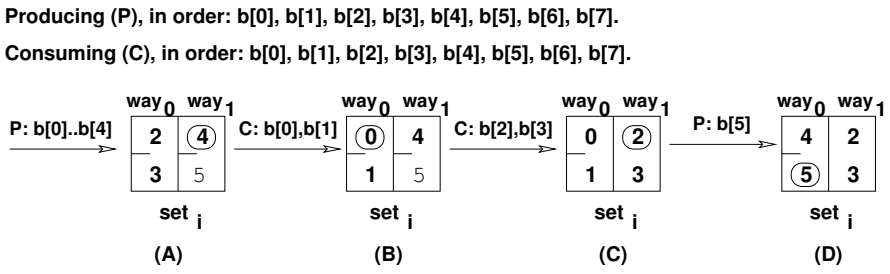

Figure 4.3: Example: access to common region - shared ways

In an associativity based partitioned cache, for the same execution sce-

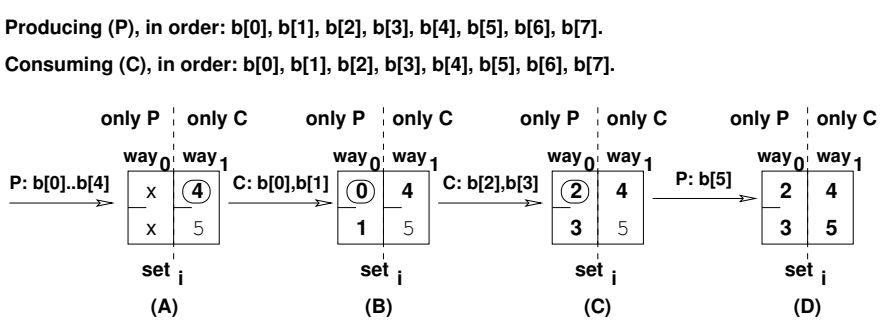

Figure 4.4: Example: access to common region - partitioned ways

nario, the consumer is not flushing the producers cache (as depicted in Figure 4.4), so when T_P accesses *b[5]* it does not miss. In conclusion, the partitioned cache ensures a hit for the accesses that exhibit spatial locality, regardless of the tasks speeds and task scheduling.

The presented example was chosen to be simple for clarity reasons. In many real situations the FIFO buffers are small enough to fully fit in the cache, but the frame buffers, for example, are not. For instance, a state-of-the art high definition image is $\approx$ 2MBytes. Common video algorithms might need to access few such image buffers [76], [60] in the same time. In this case the required picture buffers are too large to fit in a modern cache, even if the cache is fully reserved for this purpose. Fortunately, media applications exhibit high spacial locality because they process data on a block based fashion [24], [32], [99]. The typical manner to manage the access to large data instances is to exploit the spacial locality and to prefetch the blocks likely to be accessed in the near future [24], [31], [75], [116]. However, prefetching has a drawback: useful data might be kicked out of the cache to make space for the prefetched data (the so called "cache pollution" phenomenon). For large data instances, our scheme ensures full exploitation of the spatial locality and offers support for prefetching without the cost of polluting other tasks' cache. The cache partition allocated to one task acts as an "active window" over the common region. Depending on the life-time of the stored data, different cache allocated size might give result in different performance. Hence, when determining the cache size, the life-time of the data is important.

When we talk about inter-tasks shared data, their life-time is related with tasks scheduling. Thus from the temporal locality point of view, the inter-task data reuse is dependent on the scheduling and tasks speeds (i.e. execution

times). In the example in Figure 4.3 the consumer would have a cache hit if it would access a cached *b* element before the producer write another element in the same cache location (and swaps out the old *b* element). A tasks schedule in which the consumption of a cache *b* element occurs before it is flushed out of the cache by the production of another *b* element (only hits) is in principle possible because the cache of *b* is reserved only for the producer and the consumer. The problem of determining this schedule and the corresponding cache window size (or determine this schedule under window size constraints) resembles a scheduling with memory constraints problem, formulated in general as follows: determine the tasks start times and the storage element sizes such that some performance criterion (throughput, latency, etc.) is optimized or guaranteed. This problem type is mostly solved for applications represented by different variants of Synchronous Data Flow (SDF) graphs (finding the buffers' sizes problem) [117], [9], [61], [96]. In order to solve this problem the detailed task timing and synchronization has to be known at design-time, which is the case for SDF representations. However, in this dissertation we concentrate on achieving spatial compositionality, thus the tasks scheduling policy is not our subject. Moreover, on the platform we consider the exact timing of each tasks it is even not known at design-time when the static cache partitioning is decided. In our approach we take advantage of the fact that due to cache separation task analysis (thus also the one that might be required for prefetching) becomes local to every task, therefore knowledge about run-time inter-task scheduling is not needed. Based on this property, in Section 4.4 we present an off-line method of determining the cache size for the common regions via simulation.

For our desired purpose, i.e., compositionality, all the tasks that access a common region should have each at least one way of the shared cache sets assigned to the common region. Thus for this scheme to be implementable, the cache associativity should be greater or equal with the number of tasks sharing the common region. In our example, the cache set shared by the producer and the consumer should have at least two ways, one for every task. We note here however that in our case this is in principle not an issue as for multimedia applications the maximum number of tasks that share a common region is typically smaller than the number of tasks forming an application. In the extreme case when the number of task sharing a common region is larger than the associativity, there are not enough ways to exclusively allocate at least one to each task. Thus, some tasks might have to share the same way. In general, the interference among tasks that share a piece of cache can be of two types:

- caused by task switching. On a processor task execute one by one (i.e.,

in a time multiplexed fashion). The cache content of a task that is swapped out can be flushed out by the task that is swapped in. This interference is also present for a single processor executing multiple tasks, so we call it ”single processor cache trashing”.

- caused by tasks running concomitantly. This interference is present only in multi-processor systems, so we call it ”multi-processor cache trashing”. Note that in a multi-processor both types of cache trashing occur in an orthogonal manner.

There are two ways to predict the application behavior in a system potentially exposed to cache interference: (1) predict an upper bound on the cache interference effects on performance, and (2) completely ban the interference (e.g., by partitioning). The first method is available for predicting the single processor cache trashing, also known in the literature as Cache Related Preemption Delay (CRPD). Multiple solutions for determining the performance effects of CRPD already exist [81], [48], [107], [111]. In [81] and [111] CRPD is estimated for instruction caches. In [48] and [107] the case of data caches is also tackled. Hence in single processor systems, even though the tasks may interfere in cache, the performance can be predicted because the interference overhead can be calculated.

In the case of multi-processors, for tasks that share a processor, single processor cache trashing occurs, and CRPD methods can be applied. Thus in order to predict the performance of the multi-processor we only have to take care of the interference among tasks that run on different processors. The maximum number of tasks that can simultaneously run on different processors is equal to the number of processors in the system. In order to ban the cache interference by partitioning, thus compute the system performance, the cache associativity should be greater or equal with the number of processors. In particular for a common region, the number of processors that access that region should be smaller than or equal to the cache the associativity.

4.3 Mixed cache partitioning implementation

In Section 3.2 implementation schemes for cache partitioned per task basis are presented. In this chapter we propose to allocate exclusive cache sets also to common regions. Therefore each memory access should be labeled with a *task_id* or a *comm_reg_id*. The *task_id* for every processor is stored in a register and updated at every task switch, therefore it can be used directly. In the following we present the options to obtain a *comm_reg_id* first for data and

then for code.

There are several ways to obtain an *id* for the common task data. The first option is to use a *comm_reg_id* register, and to change the compiler such that it becomes aware of which variables are shared, in order insert in the code the instructions that keep this register up to date. This options restricts the way an application can use memory and pointers to memory, in the sense that all shared variables and their data size should be known at compile time. Alternatively, a part of the address can be used to encode the *comm_reg_id*. This approach requires a cache aware memory allocator, reduces the usable address space (fragmentation), and also requires adapting the compiler for handling shared static data structures. Nevertheless, for dynamic memory allocation the partitioning can be implemented relatively straightforward by providing a dedicated allocation primitive for shared buffers. A third approach is to provide the applications with primitives to register a shared memory region and to keep a table with intervals of shared memory. Moreover, for every access the cache can lookup if the address has an valid associated *comm_reg_id*. The index translation is dictated by the *comm_reg_id* if valid, or otherwise by the *task_id*, as the access is private to the task's data. In practice, this third approach is more expensive in terms of area and power. However, for our experiments we choose this last alternative because we are mainly interested in the system level aspects (e.g., inducing the compositionality, implication in miss rate). The third approach is more generic than the others because any address range can be placed in any cache region. This easily allows for other experiments, like for example separating tasks' instructions and static variables in the cache or sharing some cache partitions.

Similar to the manner we get a *comm_reg_id* for data, we could also obtain such an *id* for instructions. However this would require the compiler to register the shared code regions (functions), thus to do extra analysis to determine which function is called by more than one task. Moreover, such an approach requires that the called functions three of a task to be completely specified at design-time. Instead of this rather intricate and restrictive method we consider as shared all the instructions of tasks with the same functionality. For example, for the PiPTV introduced in Chapter 2, both idct tasks are considered to share all their instructions. The tasks with the same functionality instantiated by an application are known at compile time, thus this approach requires no extra analysis. We extend the list of *task_id* with *task_code_id*, that is used for the code of each task. For a set of tasks that share instructions, the $MASK$ and $BASE$ are the same, leading to a shared instruction partition for those tasks. Having a set of *task_code_id* requires to distinguishes between the access to

code or to data. We realize this by using an extra bit to label the L2 accesses coming from the L1 instruction cache as 'code'. This method does not guarantee that in some cases, some code is not duplicated in cache, but as the code is read only, this does not represent a problem for the correct execution of the applications.

In this chapter we provide an extension mechanism such that the partitioning methods described in Chapter 3 works not only for tasks, but also for common regions. Moreover, we realize the mixed partitioning by implementing set and associativity based partitioning according to the guidelines presented in the previous chapter. In the following we present two methods to compute the cache partitioning ratio such that (1) the number of misses or (2) the throughput of an application is enhanced.

4.4 Cache partitioning ratio

As already mentioned, we assume that an application A is composed out of N tasks, $\mathcal{T} = \{T_i\}_{(i=1,N)}$ and M common regions instances $\mathcal{CR} = \{CR_j\}_{(j=1,M)}$. We extend the sets $\{c_i\}_{(i=1,N)}$ and $\{m_i^k\}_{(i=1,N;k=1,C)}$ representing task's T_i assigned cache size and number of misses function of cache size k, with values corresponding to the common regions. Thus, we now have $\{c_i\}_{(i=1,N+M)}$ and $\{m_i^k\}_{(i=1,N+M;k=1,C)}$ where the values $\{c_i\}_{(i=1,N)}$ and $\{m_i^k\}_{(i=1,N;k=1,C)}$ correspond to the tasks and $\{c_i\}_{(i=N+1,N+M)}$ and $\{m_i^k\}_{(i=N+1,N+M;k=1,C)}$ to the common region CR_j's with $j = 1, 2, ..., M$. Furthermore, we denote with sh_j the set of tasks that access the common region CR_j ($sh_j = \{T_i | T_i \ accesses \ CR_j\}$). Moreover, w_j^i represents the number of ways the task $T_i \in sh_j$ has allocated in the CR_j cache sets.

The objective is to find the number of cache sets c_i for every task T_i and every common region CR_j and the number of ways w_j^i for every CR_j and $T_i \in sh_j$ such that a certain criterion is optimized. In the following subsections we present two methods to find the cache partitioning ratio. The first method minimizes the total application's number of misses. This is an extension to the method in Section 3.4, such that common regions are also supported. The second method maximizes the application's throughput.

4.4.1 Misses minimization

When the cache is exclusively partitioned among tasks and common regions, the number of times a task accesses a common region is dictated by the task's code and its input data, and it is independent on the cache allocated for the private task's data. The misses experienced in a common region's cache depend on the number of accesses to that region and their pattern. As these accesses are not influenced by the private task's cache sizes, the misses in a common region's cache can be optimized independently of the misses in the private tasks' cache. In other words, finding c_i and w_j^i are two independent problems, under the assumption of compositionality and exclusive cache partitioning. The mixed partitioning method proposed in Section 4.2 has two steps: (1) set based partitioning among tasks and common regions, and (2) associativity based partitioning inside common region cache. For both steps the fundamental optimization problem is the same: how to partition a group of resources among a set of entities, such that the combined performance of all the entities is enhanced. In Section 3.4, this problem was instantiated and solved for the case of assigning cache sets to tasks. We separate the problem of cache allocation in this section into two: first inside each common region (finding w_j^i) and second among tasks and common regions (finding c_i). We solve both these allocation problems utilizing an algorithm similar to Algorithm 1 introduced in Section 3.4.

To compute the associativity based partitioning ratio inside a common region (i.e. the set of $\{w_j^i\}_{(j=1,M;T_i \in sh_j)}$) we utilize an algorithm like Algorithm 1 in Section 3.4 but with the following substitutions: (1) instead of a set, the basic allocable cache unit of a task is a way. Unlike with sets, a task may have a number of ways w_j^i that is not a power of two. (2) only the set of tasks accessing the common region ($T_i \in sh_j$) participate in the optimization process for that region and (3) instead of the total cache size C, the cache limit is the number of cache sets c_{N+j}, allocated to region CR_j. Hence for each common region we have to solve such an optimization problem. An instance of this problem has to be solved for every possible value of c_{N+j}. Because of implementation reasons presented in Chapter 2, the c_{N+j} sizes are actually limited to power of two, therefore the number of possible c_{N+j} is not large.

Further, to determine the ratio for the set based partitioning, not only tasks, but also common regions have to be taken into account. The objective of the *Cache Allocation Problem* is to find the size of cache c_i for each task T_i and each common regions CR_j, such that the overall number of cache misses is minimized. The number of misses inside a common region, m_j^k is actually the

smaller number of misses obtained from the optimization process, as described in the previous paragraph. As a consequence the algorithm for set based partition has *N+M* main steps, iteratively adding to the solution all the tasks as well as all the common regions.

Given that we presented two methods to find both set and associativity based partitioning ratio and taking into account the fact that the corresponding optimization problems are independent, we can conclude that we offered a method to find the partitioning ratio that optimizes the application number of misses, for the task centric cache management.

4.4.2 Throughput maximization

Media applications typically have to be process a certain amount of data before a time deadline. For such an application, the throughput is defined as being the amount of data units processed in a time unit (for example, real time video decoding may require 30 frames per second). As we consider applications consisting of a graph of communicating tasks the throughput of such an application is bounded by the longest path in the task graph that has to be executed sequentially due to data dependencies (the critical path). Minimizing the overall number of misses, does not necessarily minimize the critical path length (improve the application throughput).

We denote with E_A the time needed by the application A to process a given number of data units that is relevant for A's utilization. Then, the throughput of the application (Th_A) is the inverse of the execution time needed to process a given number of data units: $Th_A = \frac{1}{E_A}$.

For clarity reasons, in the section we make the following notation: the cache sizes allocated to tasks T_i and common regions CR_j (representing the cache partitioning ratio) are shortly denoted with $CPR = \{c_i\}_{(i=1,N+M)}$. As mentioned already, due to implementation efficiency reasons, the cache sizes have to be a power of two number of cache sets.

In this section we tackle the problem of finding the partitioning ratio *CPR* that gives the best throughput. This problem is a *Cache Allocation Problem*, as defined in the previous chapter in Section 3.4. In Section 3.4 we proved that the $\mathcal{CAP}$ for minimizing the application's number of misses is an NP-hard problem. Intuitively, the $\mathcal{CAP}$ for maximizing the application's throughput is also a partitioning problem, as the $\mathcal{CAP}$ for minimizing the application's number of misses, thus they are similar in hardness ($\mathcal{CAP}$ for minimizing the application's number of misses is an NP-hard problem). In addition, a formula

that computes the application's throughput from the tasks' throughput is more complex than the summation that relates an application's number of misses to the tasks' number of misses. Because the tasks that compose an application can have complex interdependencies, and some of them are executed in parallel, we cannot write something like: $Th_A = \sum_{i=1}^{N} Th_i$, where Th_i is the throughput of task T_i. Thus the Dynamic Programming solution presented in Chapter 3 is not applicable for the $\mathcal{CAP}$ that has as objective throughput maximization. Therefore in the following we present an heuristic for solving this problem.

Such a problem has a large solution space therefore the searching process is time consuming. Simulated Annealing (*SA*) [59] is a well-known, powerful technique for combinatorial optimization problems, like for instance resource partitioning. The advantages of this method is that falling into a local optimum is less probable than with Greedy-like solution space searching methods. A Greedy method has as intermediate solutions only points that are "better" than the solutions already founded, therefore when the solution space is not monotonic, the method might converge toward a local optimum. As *SA* accepts some intermediate solutions that are not necessarily better than the solutions already found, that might guide the solution out of a local optimum, thus there is a larger possibility to find the global optimum. Because of this property, we use simulated annealing (as formulated in the next subsection) to solve the throughput optimization problem.

At each and every step of the *SA* optimization, the throughput of the system has to be estimated in order to decide if the current partitioning ratio is a potential optimum candidate. An analytical formula that combines the execution time of every task to determine the throughput is available for very restricted, systems. As we have chosen for flexibility and natural load balancing (therefore the scheduling policy is dynamic), the throughput cannot be analytically formulated. Therefore a simulation is required in order to obtain the throughput value needed for the *SA* evaluation stage.

An usual simulation of the multiprocessor platform is accurate, but slow. In order to find the best throughput, the *SA* process has to performs many steps, so if we would use the regular multiprocessor simulation the problem would be unsolvable in a reasonable time. Instead of a regular simulation, we use a fast, "light" simulation of the system. This means that only the FIFO reads and writes are simulated to ensure inter-task synchronization, whereas the rest of the instructions are only accounted for their execution time.

In the following we present in detail the proposed *SA* optimization method.

Simulated annealing

The simulated annealing optimization process used in this section has the followings stages:

- *Initialization.* During this phase the temperature is set to a high value, Θ_0. The current partitioning ratio CPR_{curr} is initially set to a random value, CPR_0. The throughput of the application Th_{curr} corresponding to CPR_0 is determined using the light simulation method introduced next, in Subsection 4.4.2.

- *Cooling*. This stage together with the next one (evaluation) are at the core of the optimization process and they are iteratively performed. At every cooling iteration a new solution candidate CPR_{new} is proposed. This candidate partitioning ratio is generated by making a change in the current partitioning ratio CPR_{curr}. The CPR_{curr} change is realized by halving or doubling the cache sets of a random task or common region. We allow only halving or doubling because the cache sizes should be a power of two number of sets due to implementation efficiency reasons. The available cache can be exceeded in the case of doubling the cache sets of a task. In order to increase the chance of finding a global optimum, for a cooling step we tolerate ΔC more cache sets over the available value C.

  At the iteration k of the cooling stage the temperature T_k decreases according to the formula: $\Theta_k = \alpha \cdot \Theta_{k-1}$, where α is a given constant, smaller than 1.

- *Evaluation*. In this *SA* stage it is decided if the current partitioning ratio CPR_{curr} is updated to CPR_{new}. For this, the throughput of the system Th_{new} is determined using the already mentioned light simulation method. If the difference in throughput is $\Delta Th = Th_{curr} - Th_{new}$, the new ratio becomes the current ratio according to the following probability function:

$$p(\Delta Th) = \begin{cases} e^{\left(-\frac{\Delta Th}{\Theta}\right)} & (\Delta Th > 0) \\ 1 & (\Delta Th \leq 0) \end{cases} \tag{4.1}$$

  This means that if the new throughput is larger than the current throughput, the current ratio is updated to the candidate ratio. Otherwise, if the new throughput is smaller than the current throughput, there is still a non zero probability that the new candidate solution is accepted in order to

increase the chance of finding a global optimum and not "falling" into a local one. However, as Equation (4.1) suggests, if the temperature cools down, the chance of accepting a worse candidate is diminishing.

- *Termination*. The *SA* optimization terminates if the temperature is zero or if the optimum is not changed for *I* iterations. The final solution is the partitioning ratio *CPR* corresponding to the largest throughput *Th*, that respects the constraint that the total allocated cache is smaller than or equal to the available cache *C*.

Light simulation

At each *SA* evaluation step, the application throughput has to be estimated. As a regular simulation is unacceptable due to its low speed, and a throughput formula is not available for flexible systems like ours, in this subsection we propose a fast, light simulation strategy.

The main idea of the light simulation strategy is that only the inter-task synchronization is simulated, whereas the rest of the instructions are only accounted for their execution time. In our case the synchronization operations consist of *read*s/*write*s from/into a blocking FIFO, thus in the following we discuss only this synchronization type. We note however, that the light simulation is not restricted to FIFO-based synchronization. Other types of synchronization can be easily supported, in a similar manner as the FIFO-based one. The execution times are obtained via an initial limited set of regular simulations (i.e. each task is simulated once with every possible cache size it can have allocated). The process is detailed at the end of this subsection, as first we introduce the light simulation method.

To explain how light simulation works we first discuss a simple example with two communicating tasks as depicted in Figure 4.5(A) and then we detail this method for the general case. The task T_P is the producer of data in FIFO F_0 and the task T_C is the consumer of these data. Figure 4.5(B) depicts the code for the two tasks in a C-like language. On one hand the producer T_P is computing data *x* and then writing it in the FIFO F_0, with the granularity of one unit. On the other hand the consumer T_C is reading the data *y* from the FIFO F_0, with a granularity of *3* units, and then T_C is using *y* for some computation (the exact detail of this computations are not relevant for the light simulation strategy). For both tasks these operations are performed multiple times, in a loop.

In order to perform the light simulation one has to derive the execution

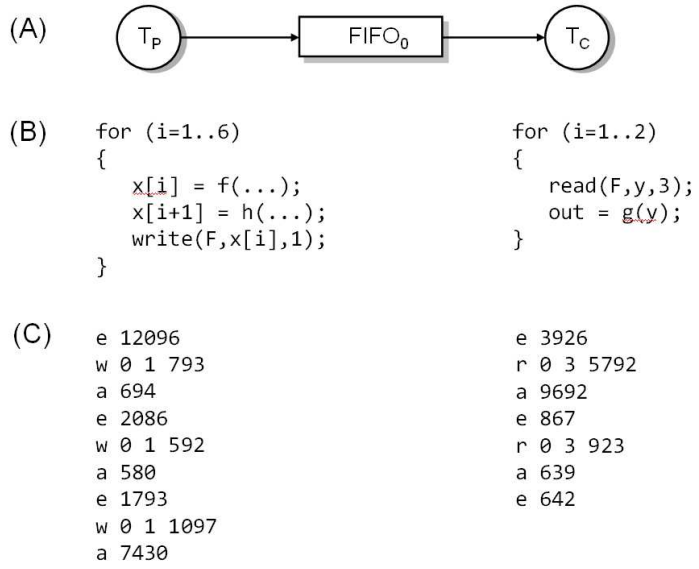

Figure 4.5: Light simulation example: producer-consumer

times and the FIFO read/write sequences of the two tasks out of the tasks' code. A task trace is the timed list of "*execute*" (*e*), "*read from FIFO*" (*r*), "*write to FIFO*" (*w*), and "*access to common region*" (*a*) actions. The tasks code presented in Figure 4.5(B) has the corresponding traces described in Figure 4.5(C). The task T_P trace starts with an *execute* action corresponding to the computation of data x. This *e* action has associated the time spend by T_P in the computation of *x* (*12096* cycles in our case). This computation is not simulated, but its execution time is added on behalf of T_P. Then T_P performs a *write* action that has associated the FIFO id *0*, and the number of send tokens equals 1. This *w* action is simulated to ensure proper inter-task synchronization. Then the task T_P accesses the shared data region corresponding with the FIFO F_0. To represent this we use an *a* action that has associated the time spend in accessing these shared data (*694* cycles in our case). Again, just like for the *e* action, this time is not simulated, but only accounted for T_P. Then this sequence of actions repeats, the trace being actually an unrolling of the *for* loop. Similarity, the trace of the consumer task T_C, has a sequence of *e* actions, this time *read*, not *write* actions and *a* actions. The *r* action has associated the FIFO id *0* and the number of tokens received (3 in this case). Like for the task T_P, the synchronization actions (*r* in this case) are the only one simulated,

while the *e* and *a* actions being just accounted for their time on behalf of T_C.

In the general case, the light simulator interprets the traces of *N* tasks running on *P* processors. The simulator has a notion of "processors", in the sense that the number of task that can be simultaneously active is smaller or equal than the number of available processors, *P*. In the following we describe how the simulator interprets each possible type of action a trace may contain.

The first action that we detail is the *e* action. An *e* action is behaviorally similar with a System C *"wait"* statement [105], therefore the simulator just counts the time, and immediately jumps to the next action (the time associated to an *e* action depends on the size of the cache part allocated to the task). This time counting strategy applies also to the other actions. For them the time is first counted, and then the operation is executed.

Another action type that can be encountered for an active task is the synchronization action type. The *r/w* actions have associated the FIFO id involved in the read or write operation, the number of tokens consumed or produced, respectively, and the time spend to execute this operation. This information are used to actually execute the FIFO reads and writes when encountered in an active task trace. The execution of an *r* or *w* action might result in blocking the task, if the requested tokens are not available yet. When a task blocks (becomes inactive), a processor becomes free, so other tasks may get active (its traces may be interpreted). In this manner, the light simulator imposes the same inter-task schedule as the one in the regular simulation.

The last possible type of action, the access of common region action, is treated separately than the *e* action, because its access time depends on the cache allocated to the common region, not on the task itself. However, from the point of view of the simulator an *a* action is similar to an *e* action and its associated time is just added to time of the task executing it.

To gather the traces and the execution times corresponding to each action, each and every task T_i is accurately simulated with the list of T_i's possible cache sizes. The cache sizes have to be a power of two number of cache sets, therefore, the following are the possible cache sizes corresponding to task T_i: $2^0, 2^1, 2^2, ..., 2^{k(i)}$, where $k(i) \in N$ and $2^{k(i)}$ gives the maximum cache value for task T_i. This $2^{k(i)}$ is chosen such that if the tasks has $2^{k(i)+1}$ cache sets, no changes in its performance can be observed. The trace of a task T_i contains the execution time information of that task having a certain amount of cache, or in other words, the execution time for a (T_i, c_i) pair. In the case of common regions access, for each task T_i that accesses a common buffer CR_j we collect the access times corresponding to the (T_i, CR_j, c_j) tuples, where

c_j is the cache allocated to buffer CR_j. As implied by the compositionality property, the tasks don't influence each other, therefore the execution time for a (T_i, c_i) pair (or (T_i, CR_j, c_j) tuple) can be used in every combination with the rest of the tasks. Hence, the throughput of a potential cache portion candidate $\{c_0, c_1, ..., c_{N+M-1}\}$ can be determined by a light simulation of the corresponding (T_i, c_i) and (T_i, CR_j, c_j) traces.

As we already mentioned, the traces are gathered by means of regular simulation. These traces are dumped not by the applications, but directly by the hardware simulator, such that the time involved in trace file writing does not count in the times spend for *e* or *a* actions. In detail, the manner in which the regular simulator gathers each of the *e*, *w*, *r* or *a* parameters is as follows. The read and write software calls are profiled to dump their parameters for the corresponding *r/w* actions on a special memory location. The penalty involved in tracing reads and writes equals the time to access this memory location (which on a CAKE platform is 10 cycles), therefore is negligible when compared with the rest of the computation time. The *a* actions are easy to identify (and their access time measured) because the common regions accesses are automatically detected by the hardware as the cache partitioning implementation actually requires (see Section 4.3). The rest of the time between two consecutive reads and/or writes, which is not spend in an access to a common region, simply represents the time spend in *e* action. The light simulator is implemented using the CASSE tool chain [93], which is a System C [105] based tool.

4.5 Experimental results

In this section we present the results of applying the proposed cache partitioning technique on a CAKE multi-processor platform, with 4 Trimedia processor cores and various L2 cache sizes. We use the cache partitioning described in this chapter. The experimental workload consists of two video multi-tasking applications: an H.264 decoder and a picture-in-picture-TV (PiPTV) decoder. In these experiments the L2 is four ways associative, as none of the common regions is shared by more than four tasks, thus four ways suffice for the mixed partitioning inside the common regions cache (and for the proof of concept for our methods). Both applications, being research vehicles, are not optimized for low tasks switching, thus they have a large number of tasks (and common regions). Concretely, the PiPTV application has 49 tasks and 328 common regions, and the H.264 application has 20 tasks and 111 common regions. In order to ensure compositionality, the number of available sets in the L2 should

		512K/4	1M/4	2M/4	4M/4	8M/4
PiPTV	shared	n.a	85%	83%	83%	79%
	mixed part	n.a	0.34%	0.08%	0.27%	0.20%
H.264	shared	78%	81%	77%	70%	68%
	mixed part	0.12%	0.06%	0.07%	0.04%	0.08%
$Average$	shared	78%	83%	80%	77%	74%
	mixed part	0.12%	0.20%	0.08%	0.16%	0.14%

Table 4.1: Relative inter-task conflict misses function of cache dimensions.

be larger than the sum of the number of tasks and the number of common regions of an application. As a result, for a 4 ways associative L2 with a 512 KB line size, the minimum possible L2 size for the PiPTV is 1MB and for the H.264 is 512KB.

Both applications exhibit mixed data and functional parallelism and are separately simulated on the CAKE platform. For these applications the maximum amount of tasks that share a common region is two. Therefore our 4 ways associative L2 is enough to ensure compositionality. The only exception are the data structures of the OS that are shared by all tasks. As mentioned in the previous chapter the size of these data is small so they can be fully cached in an L2 partition.

4.5.1 Compositionality

To evaluate system's compositionality we use the number of conflict misses metric *CM(A)* defined in the previous chapter, in Section 3.5. We simulate on the CAKE platform each of the two applications with each considered L2 size, with an L2 partitioning ratio corresponding to the minimum application's number of misses obtained with the method in Subsection 4.4.1 and we record the number of conflict misses. The number of conflict misses of each of the two applications are reported in Tables 4.1 and Figure 4.6. The conflict misses values of an application are presented in this subsection as a relative percentage from the application's number of misses. The last row represents the average over the two applications number of conflict misses. As the amount of cache allocated to a task has no influence on compositionality, we chose the L2 partitioning ratio such that the application's number of misses is minimized.

The experiments point out that in the case of the partitioned L2 the relative number of conflict misses is under 1% for each application in every cache combination. The existing conflict misses are due to the fact that the L2 sets of the OS's data structures are shared among tasks and not partitioned based on

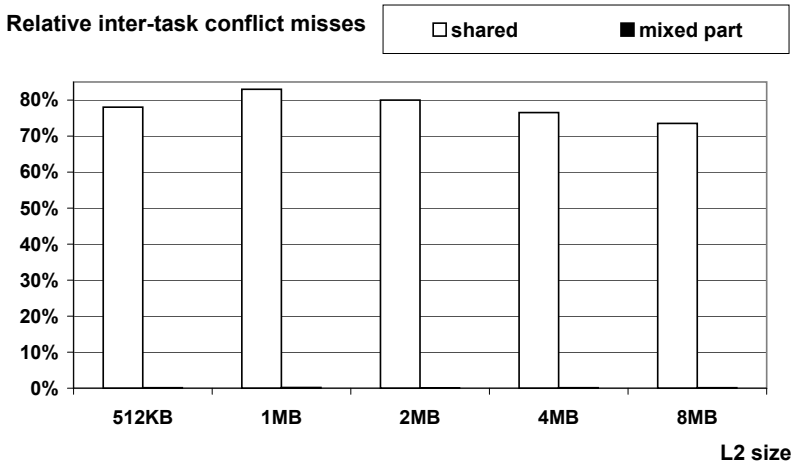

Figure 4.6: Average conflict misses: shared vs. partitioned cache

associativity (under the conditions mentioned in the beginning of this section). In the case of the shared L2 cache, a large fraction of the misses represent actually inter-task conflict misses. The peak value for these misses is 85% and the average over the two applications and all cache configuration is 78%. As also observed in Chapter 3, the number of conflict misses tends to decrease with the increase of the L2 size. This tendency is explained by the decrease of all the misses when the cache size is increased.

4.5.2 Performance

Misses minimization

In this subsection we investigate the performance implications of mixed partitioning in combination with the strategy for minimizing the number of misses. For this we simulate each of the two applications on the CAKE platform with each considered L2 size, with the following cache configurations: (1) the shared cache, and (2) the partitioned cache as proposed in this chapter, with the partitioning ratio optimized for overall least number of misses and (3) an infinite cache, in order to give an idea about the maximum possible improvement. For these cases we compare the average (over the four processors) for the number of L2 misses per instruction and the average number of cycles per instruction. As in the previous chapter, we approximate an infinite L2 with a cache of 8 Mega Bytes (MB), as the number of misses in this case is very

low and by making it larger we do not observe any substantial reduction of them. The average MPI and CPI can be viewed in the Tables 4.2 and 4.3 and in Figures 4.7 and 4.8 for the PiPTV application. For the H.264 application the average MPI and CPI are presented in Tables 4.5 and 4.4 and in Figures 4.9 and 4.10.

	1MB	2MB	4MB	8MB
shared	1.33	1.20	1.12	1.11
partitioned	1.40	1.15	1.12	1.11

Table 4.2: PiPTV's CPI: shared vs. partitioned L2.

For both applications one can observe that, for the smallest exercised L2 size the mixed partitioning degrades the application's performance. Compared with a 512KB shared cache, when having a partitioned cache of the same size, the H.264 application exhibits 10% larger MPI leading to a 6% CPI increase. In the case of PiPTV executing with its smallest cache (1MB), the experienced increase in MPI is 24% and in CPI is 5% (when compared with an shared L2 of the same size).

For the rest of the considered L2 sizes, the partitioned cache outperforms or it is at least as good as the shared cache. For the H.264, when the employed L2 has 1MB the MPI decreases with 19% resulting in a 4% CPI improvement. This CPI improvement represents 47% from the maximum improvement achievable when the L2 has an infinite size. When the H.264 uses a cache larger than 1MB the H.264's performance differences among the shared and the partitioned cases are very little (the CPI is the same and the MPI's

	1MB	2MB	4MB	8MB
shared	0.00046	0.00021	0.00010	0.00001
partitioned	0.00061	0.00013	0.00009	0.00001

Table 4.3: PiPTV's MPI: shared vs. partitioned L2.

	512KB	1MB	2MB	4MB	8MB
shared	2.34	2.01	1.88	1.88	1.84
partitioned	2.47	1.93	1.87	1.87	1.84

Table 4.4: H.264's CPI: shared vs. partitioned L2.

	512KB	1MB	2MB	4MB	8MB
shared	0.00776	0.00464	0.00154	0.00096	0.00001
partitioned	0.00861	0.00389	0.00153	0.00098	0.00001

Table 4.5: H.264's MPI: shared vs. partitioned L2.

variations are under 2%). In the PiPTV case, when the employed L2 has 2MB the MPI decreases with 38% resulting in a 5% CPI improvement. This CPI improvement represents 55% from the maximum improvement possible with an infinite size L2. When using a cache larger than 2MB the performance differences among the shared and the partitioned cases are very little (the CPI remains the same and the MPI's variations are under 6%).

As already mentioned in Chapter 3, there are two phenomena that determine the difference in misses' number between a shared and a partitioned cache. If the cache is partitioned, the inter-task cache interference is eliminated (therefore the number of misses may decrease) but each task can use less cache space than in the shared case (therefore the number of misses may increase). In our examples one can observe that for a small L2 size the second effect is dominant, whereas for larger L2s the elimination of inter-task flushing via partitioning leads to performance improvement. This behavior is intuitively explained by the fact that, in general, sharing can be beneficial when having scarce resources (e.g., a small L2). Furthermore, the number of processors is small compared with the number of tasks, and at a given moment the utilized cache fraction is the one allocated to the tasks that execute in that moment. In our static partitioning scheme the task allocated to the rest of the tasks is left unutilized. Both applications have a large number of tasks and communication buffers so the cache fragmentation induced by partitioning is relatively high. This means that each task may have a relatively small cache part, thus the cache space a task might utilize is significantly smaller in the partitioned case when compared with the shared case.

Throughput maximization

In this subsection we first present the results of applying the throughput optimization method on a CAKE platform, and then we evaluate and discuss the accuracy of the light-weighted simulation used in the *SA* optimization. Like in the previous section, we simulate each of the applications on the platform with each of the considered cache sizes, using the partitioning ratio calculated

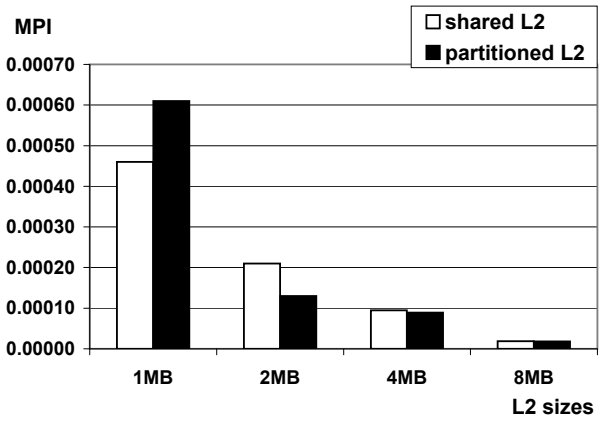

Figure 4.7: PiPTV CPI: shared vs. partitioned cache

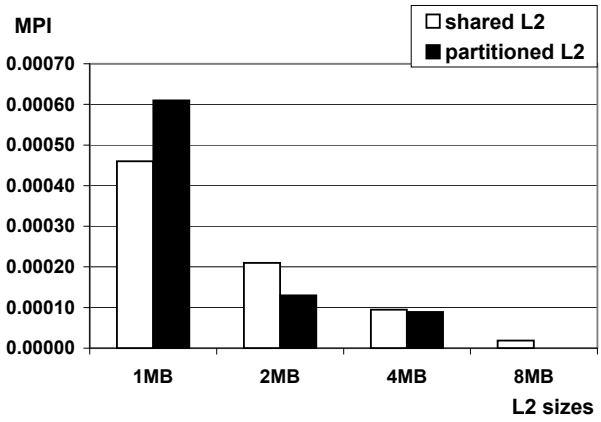

Figure 4.8: PiPTV MPI: shared vs. partitioned cache

as proposed in Section 4.4.2. The used parameter values of the *SA* optimization process are: initial temperature $\Theta_0 = 10^4$, the maximum cache excess ΔC = *512KB*, and the limit number of iterations without an optimum change *I=100*. The temperature decreases at every step with a parameter $\alpha = 0.9$.

In the case of throughput we use two metrics to asses the system performance. The first one is the throughput as defined in Section 4.4.2 (for a given number of frames equal with 25). The second metric is the average MPI and this is important because it indicates the price to pay for a larger throughput. We compare four cache configurations: (1) the cache fully shared, (2) the cache partitioned such that the number of misses is minimized (CPR_M), (3) the cache

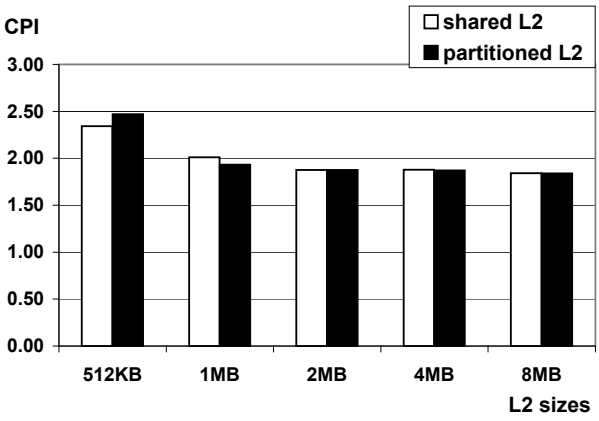

Figure 4.9: H.264 CPI: shared vs. partitioned cache

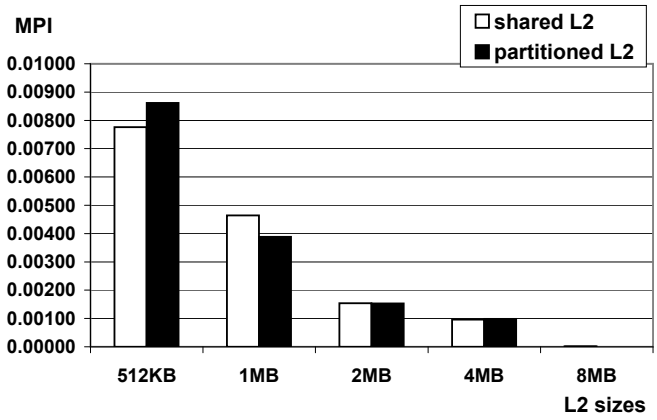

Figure 4.10: H.264 MPI: shared vs. partitioned cache

partitioned such that the throughput is maximized (CPR_T), and (4) the cache shared, but having an infinite size (approximated with an L2 of 8MB, as already specified). Table 4.8 and 4.9 and Figures 4.13 and 4.14 present the MPI and the throughput (for 25 frames) for the H.264 decoder in the four studied cache configurations, corresponding to different L2 cache sizes.

Looking at the CPR_M and the CPR_T cases, one can observe that the throughput is, as expected, larger for the CPR_T case at the expense of increased MPI. The largest difference appears in the case of the smallest investigated cache (1MB and 512KB for PiPTV and H.264, respectively), so we comment first on the results obtained with this L2 size case. On average

	1MB	2MB	4MB	8MB
shared	0.00046	0.00021	0.0001	0.00001
CPR_M	0.00061	0.00013	0.00009	0.00001
CPR_T	0.00070	0.00018	0.00011	0.00001

Table 4.6: PiPTV's MPI: shared vs. CPR_M vs. CPR_T

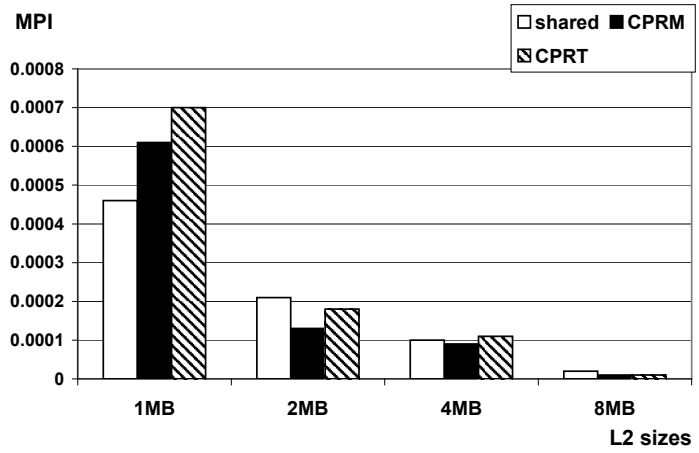

Figure 4.11: PiPTV's MPI: shared vs. CPR_M vs. CPR_T

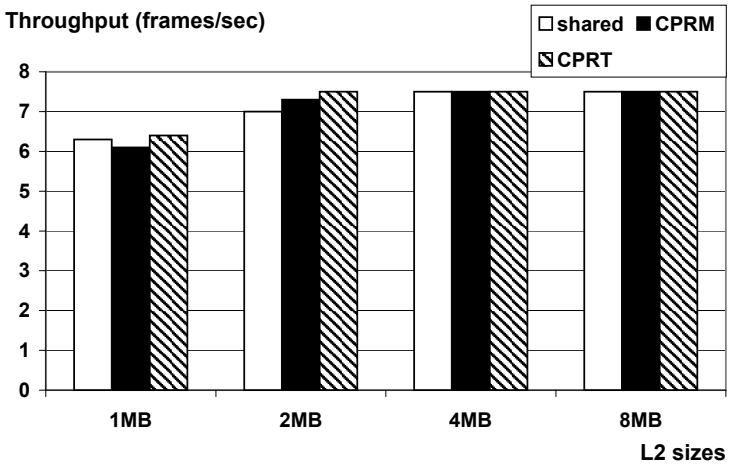

Figure 4.12: PiPTV's throughput: shared vs. CPR_M vs. CPR_T

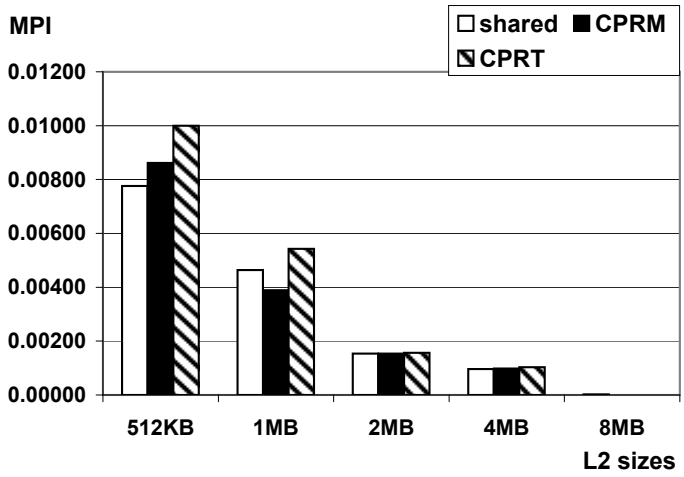

Figure 4.13: H.264's MPI: shared vs. CPR_M vs. CPR_T

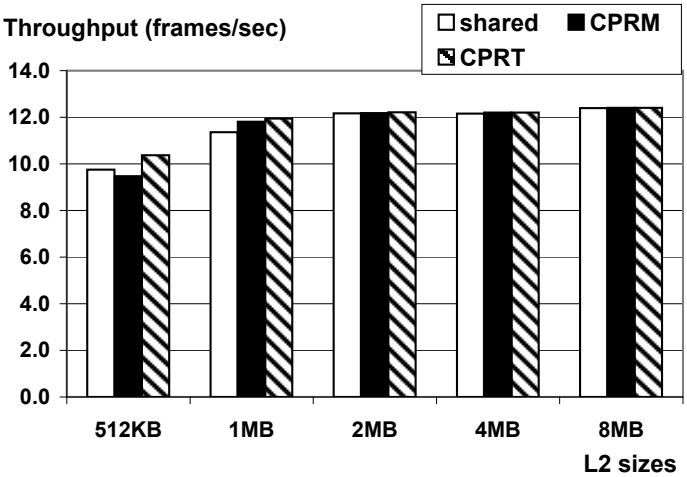

Figure 4.14: H.264's throughput: shared vs. CPR_M vs. CPR_T

	1MB	2MB	4MB	8MB
shared	6.3	7	7.5	7.5
CPR_M	6.1	7.3	7.5	7.5
CPR_T	6.4	7.4	7.5	7.5

Table 4.7: PiPTV's throughput: shared vs. CPR_M vs. CPR_T

	512KB	1MB	2MB	4MB	8MB
shared	0.00776	0.00464	0.00154	0.00096	0.00001
CPR_M	0.00861	0.00389	0.00153	0.00098	0.00001
CPR_T	0.01000	0.00542	0.00157	0.00103	0.00001

Table 4.8: H.264's MPI: shared vs. CPR_M vs. CPR_T

across the two applications, the CPR_T cache configuration has an MPI with 14% larger than the CPR_M configuration, but it delivers a 7% throughput increase. Detailed, the MPI of the CPR_T partitioned L2 is 15% and 13% higher than the one of CPR_M, leading to a 6% and 9% increase in throughput for PiPTV and H.264, respectively. This throughput improvement of the CPR_T configuration corresponds to the completion of approximately 1 extra frame per second for the H.264 decoder, and to approximately 0.3 extra frame per second for the PiPTV. When using an infinite L2 cache it can be observed that, the PiPTV has 20% throughput gain and H.264 application a 30% one, when compared with 512KB and 1MB, CPR_M. These speedups represent the maximum speedup achievable by tuning the L2 cache, and requires at least 4 times larger L2. One can observe that the proposed throughput maximization strategy brings 30% and 33% (for PiPTV and H.264) of the possible throughput improvement, while keeping a cache size that is least 4 times smaller.

For the same cache sizes as in the paragraph above, when compared to a

	512KB	1MB	2MB	4MB	8MB
shared	9.8	11.4	12.2	12.2	12.4
CPR_M	9.5	11.8	12.2	12.2	12.4
CPR_T	10.4	11.9	12.2	12.2	12.4

Table 4.9: H.264's throughput: shared vs. CPR_M vs. CPR_T

shared cache case the CPR_T cache configuration improves the throughput with 6% for the H.264, but, in return, has an MPI with 22% higher. For the PiPTV the comparison between CPR_T and a shared cache reveals that the throughput of CPR_T is close to the one of the shared cache, and the MPI is 34% larger. However, unlike the shared cache the CPR_T configuration is compositional.

As expected, the performance difference among the four cases decreases with the increase of the cache size. When the L2 is large (more than 1MB and 2MB for H.264 and PiPTV, respectively), most of the application's footprint fits in the cache, thus L2 is not a performance bottleneck, hence optimizing it does not deliver further gains.

Light simulation's accuracy

As mentioned in the Section 4.4.2, during the *SA* a light simulation of the system is performed to evaluate the throughput corresponding to a given partitioning ratio. Whereas this is certainly faster than the normal simulation, the question that remains is if the light simulation is accurate enough. Thus in this subsection we investigate the light simulation's accuracy. Figure 4.15 depicts the average completion time for the H.264 decoding of 25 frames in two cases: regular simulation and light simulation. The comparison between these two cases is presented for multiple cache partitioning ratios. The maximum difference between the completion time reported by the regular simulation and the one of the light simulation is 3%. The 3% difference is actually caused by the fact that the system is not 100% compositional. Some tasks' timings are slightly different from a configuration to another because the L1 cache is not partitioned. The L1 is considered private to each task during its execution. However, there are variations in the cache access pattern due to L1 behavior. A very important fact is that the light simulation is at least 30 times faster than the regular simulation, while being only 3% away from the precise result.

We would like to mention that the experimental results presented in this section are obtained using a rather old version of the Trimedia cores embedded on the CAKE platform. As one could notice, in our setup the H.264 and PiPTV decoding cannot be done in real time (25 or 30 decoded frames per second). However, the results in this section suggest that the proposed optimization methods can bring performance improvement. We believe that these improvements can be achieved also with newer Trimedia cores. Presently H.264 decoding of a simple definition video stream can be realized in real time on a single Trimedia 3270, whereas decoding a high definition stream would require multiple such cores. As an optimized mapping of the H.264 on the new Tri-

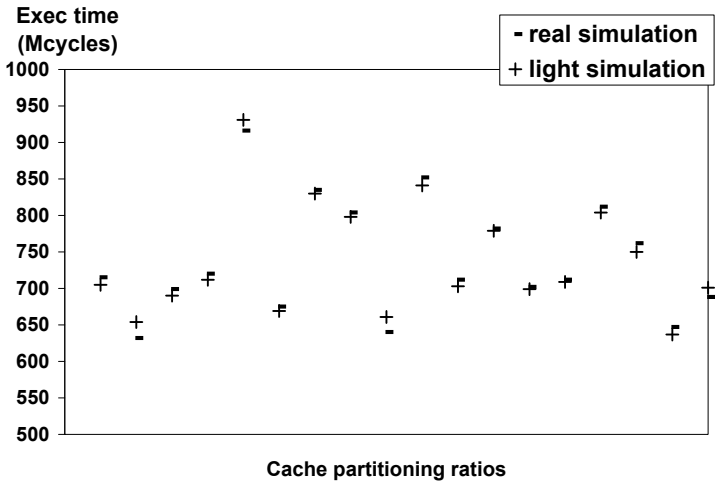

Figure 4.15: Light simulation accuracy

media core was not available at the date of this research, it is subject to future research to determine the exact potential of cache partitioning in improving the performance of such cores.

4.6 Conclusion

In this chapter we propose a method that contributes to the use of a multiprocessor with shared caches in real-time systems. We developed a set and associativity based cache partitioning technique that ensure performance compositionality within reasonable bounds and allows cache sharing for common tasks data and/or instructions.

Our method removed the inter-task cache interference by using two cache partitioning types. First, each task and each inter-task common data had allocated an exclusive part of the cache sets. Second, inside the cache sets of common data region each task accessing it had allocated a number of ways.

Subsequently, to the mixed partitioning method we propose two techniques to find the cache partitioning ratio. The first one has as purpose to minimize the overall number of misses (based on a Dynamic Programming formulation) and the second one has as purpose to optimize the throughput of the applications (based on Simulating Annealing).

The mixed partitioning method was applied to the shared L2 cache of a

CAKE multiprocessor. Two multi-tasking applications were used for the experiments: H.264 decoding and picture-in-picture-TV (PiPTV). The tasks of both applications share instructions and data. The experimental results indicate that the proposed cache partitioning ensures compositionality to a large extent. For both applications the number conflict misses represent less than 1% from the total misses.

From the performance point of view we compared the following four cases: (1) the cache fully shared, (2) the cache partitioned such that the number of misses is minimized (CPR_M), (3) the cache partitioned such that the throughput is maximized (CPR_T), and (4) the cache shared, but having a virtually infinite L2 size (a size that contain the entire application's footprint - 8MB in the present case). We measured the L2's performance (in number of L2 misses per instruction) and the application's performance (in number of cycles per instruction and throughput). As expected, the performance differences among the four cases are dependent on the L2 cache size. For small caches the fragmentation of the cache entailed by partitioning caused a performance decrease of the CPR_M when compared with the shared cache. On average over the two applications the MPI increases with 17% and the CPI with 6%. However, for these sizes the CPR_T case improves the throughput at the expense of extra misses. When compared with CPR_M, the CPR_T increases the throughput with 7% on average, under the circumstances that a maximum of 25% is actually possible by having an infinite L2 cache. Thus, while having less than a quarter of the "infinite size" L2, the proposed method achieves more than a third from the maximum throughput improvement that is possible with an L2 cache of 8MB. This throughput increase corresponds to an increase in number of L2 misses per instruction with 14%.

For average cache sizes the elimination of inter-task misses caused by partitioning supersedes the effect of cache fragmentation. The CPR_M partitioned L2 has 5% better CPI and 28% better MPI than the shared cache, on average. The CPR_T has more or less the same performance as the CPR_M. Therefore, as both *CPR* methods deliver compositionality, when a speedup is desired, for the exercised applications, in the case of small L2 sizes the CPR_T optimization is preferable, whereas for larger L2 sizes the CPR_M one should be applied.

In the case of throughput optimization, at every step of the annealing, the throughput of the system has to be estimated very fast, so we utilized a light simulation strategy. Compared with a regular simulation, the light simulation is 30 times faster and its accuracy is within 3%.

In the previous and current chapters we presented the advantages of a static

partitioned L2 cache, from compositionality and performance points of view. In the embedded domain context a subsequent question is weather the robustness of the system is affected by cache partitioning. In order to be able to guarantee performance, the designer should be able to estimate the deviations due to internal variations caused by task switching, and also due to external variations caused by different input data. Hence in the following chapter we investigate the internal and external robustness of the system in the presence of cache partitioning.

Chapter 5

Cache partitioning robustness

In this chapter, we propose a method to assess the robustness of the cache management scheme introduced in Chapters 3 and 4 , which utilizes static L2 cache partitioning to induce compositionality to the system. However, the compositionality is not 100% ensured because the L1 cache is assumed to be private to each and every task during its execution and only the L2 is partitioned. When the task switching rate is high, this is might not be a very realistic assumption. Thus, in order to guarantee performance, one should be able to estimate the variations induced by the L1 inter-task sharing. Moreover, the partitioning of the cache is a static one, thus the application may use only one partitioning ratio during its entire execution. This cache partitioning ratio is computed utilizing the application's statistics for a given input data set, as described in Section 3.4. However, during the application execution different other input data might have to be processed. It is quite probable that for these new data sets the partitioning ratio for which the application has its best performance is different than the one which is in use. In order to guarantee that a certain performance is delivered by the system, the designer should be able to estimate these deviations too.

In the view of previously mentioned phenomena two *robustness* aspects are relevant in our context: (1) the variations introduced by the inter-task L1 interference (2) the variations induced in the L2 behavior corresponding to various input data sets. The first robustness type is addressed as "intern" because the instabilities are caused by the tasks comprising the application. The second robustness type is addressed as "extern" because the variations in performance are caused by the extern input stimuli. In the following we propose an approach to assess the robustness of an application running on a multi-processor

system with statically partitioned L2 [66], [67]. As mentioned, for this type of systems the internal robustness is determined by inter-task interference in the L1 cache. This interference strongly depends on the task switching rate. To estimate the internal robustness we introduce a *sensitivity metric*, which reflects the variation in L2 misses number for different task switching rates. To assess the external robustness, we introduce the *stability metric*, which measures the performance deviations for the case when the application processes another input data set than the one utilized to determine the static partitioning ratio. For a given cache partitiong, an application is considered to be stable if its number of misses obtained with a certain input data is close to the least number of misses possible for that input data.

The outline of this chapter is as follows. In Section 5.1 we present a method to investigate the internal robustness of the proposed cache management method. In Section 5.2 we present a method to investigate the external robustness, in Section 5.3 we present the experimental results, and in Section 5.4 we draw the conclusions.

5.1 Internal robustness

In a memory organization like the one we consider, the internal variations in task performance are due to the fact that task switching pollutes the L1 caches. When, on a processor P_k, a task T_i is swapped out by a task T_j, T_i's data are gradually flushed out of P_k's L1 by T_j memory accesses. The amount of data that T_i might still find in the cache on its next execution on P_k depends on how long T_j was executed and on whether other tasks were executed in the mean time on P_k. High task switch rates are likely to pollute L1 caches less at a time, but for many times. Low task switch rates are likely to pollute the L1 cache more at a time, but rarely. The exact amount of L1 pollution depends on the application. For a picture-in-picture video decoder our experiments indicate that when the average task switching rate almost doubles (from 24K times/second to 41K times/second) the number of accesses to the L2 cache increase with 60%. Under these conditions, if a certain off-chip bandwidth has to be guaranteed to tasks or applications, the robustness of the system to task switching rate has to be investigated.

For the internal robustness analysis we propose to use the L2 sensitivity function. In order to define it, let us assume that the application is composed out of N tasks, $\mathcal{T} = \{T_i\}_{(i=1,N)}$ and that $SWR = \{swr_r\}_{(r=1,R)}$ is the set of investigated task switching rates. The number of L2 misses of task T_i depends

on T_i's allocated cache size c_i, and on the task switching rate swr_r. We denote these T_i's L2 misses with $miss_i(c_i, swr_r)$. The L2 sensitivity corresponding to a task T_i is defined as being the maximum difference in the number of L2 misses among the investigated task switching rates, when a given L2 cache size c_i is allocated to T_i. To give an idea about the impact of this variation on the application performance, we define the task sensitivity relative to the number of misses obtained when the tasks switch at a reference rate, *swr*:

$$sens_i(c_i) = \frac{\left| \max_{SWR}\{miss_i(c_i, swr_r)\} - \min_{SWR}\{miss_i(c_i, swr_r)\} \right|}{\sum_{i=1}^{N} miss_i(c_i, swr)} \times 100\%. \quad (5.1)$$

For a relevant estimation, the reference task switching rate *swr* should be the most probable, real-life, task switching rate. If this value is not known or it is variable, the application designer might choose to relate to the application misses obtained for one of the swr_r, or to an average over them.

In the same way as the task's sensitivity, we define the application's sensitivity $sens_A$ as being the relative maximum difference in overall number of misses over the investigated task switching rates, when a certain L2 partitioning ratio is applied:

$$sens_A = \max_{T_i \in T}\{sens_i(c_i)\}. \quad (5.2)$$

The smaller $sens_A$ the more robust is the application. Ideally, we would like to get $sens_A = 0$, but this cannot be achieved for the case when only L2 is partitioned. The platform we consider has also a level of L1 caches, which in this thesis are not considered subject to inter-task interference. In reality this is not the case, but, due to typical small sizes, L1 is unsuited for static partitioning. In a multi-processor system, if L1 is statically partitioned the application's tasks should be statically assigned to processors (it makes no sense to allocate cache for a task on a processor where that task might never run). This is not a preferred option because it restricts the run-time processors' load balancing options. For example in a video decoder where all tasks concur for processing frames at a certain rate, restricting run-time load balancing can diminish the performance. Even, in the case that L1 is dynamically partitioned, the application's sensitivity $sens_A$ still cannot be zero because the repartitioning is dictated at run-time, therefore variations may occur.

5.2 External robustness

This subsection presents a method to determine the performance deviations for the case when the application processes another input data set than the one utilized to determine the static cache partitioning ratio. First we illustrate the analysis of external robustness by using a small example, and after that we present the general formulation of this analysis.

Let us assume that the investigated application has three tasks ($N = 3$) and two relevant sets of input data in_1 and in_2 are considered in the cache partitioning process. Let us assume that when the application uses in_1 (in_2) as input data its best performance is achieved if tasks have as (optimal) partitioning ratio $OPR_1 = \{c_1^1, c_2^1, c_3^1\}$ ($OPR_2 = \{c_1^2, c_2^2, c_3^2\}$), as illustrated in Figure 5.1. OPR_1 and OPR_2 are calculated using the Algorithm 1 introduced in Section 3.4, such that the application's L2 misses is minimum, under the constraint that the allocated cache is smaller that the available cache (14 units in our case).

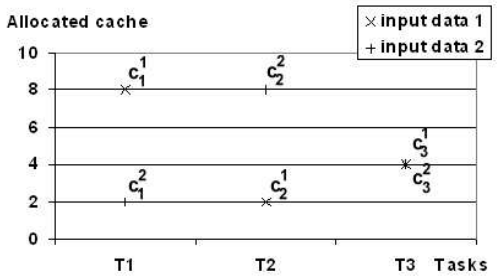

Figure 5.1: Example: Partitioning ratios corresponding to two input data

It can be observed that the best partitioning ratios OPR_1 and OPR_2 are different. When using static cache partitioning the application may use just one single partitioning ratio, $OPR = \{c_1, c_2, c_3\}$. This ratio can be OPR_1, OPR_2, or any compromise between those two. For instance any partition with $c_1 \in [\min(c_1^1, c_1^2), \max(c_1^1, c_1^2)]$, $c_2 \in [\min(c_2^1, c_2^2), \max(c_2^1, c_2^2)]$, and $c_3 = c_3^1 = c_3^2$ can be utilized.

If, for example, OPR_1 is not used as the partitioning ratio, in case the application is processing in_1 as input data, its performance is deviating from the best achievable one. In this case it is of interest to estimate an upper bound of the potential performance degradation. For this purpose, we cal-

culate the worst partitioning ratio, $\overline{OPR_1} = \{\overline{c_1^1}, \overline{c_2^1}, \overline{c_3^1}\}$, with $\overline{c_1^1}, \overline{c_2^1}$, and $\overline{c_3^1}$ bounded by OPR_1 and OPR_2. $\overline{OPR_1}$ is determined utilizing the same Dynamic Programming algorithm, as for OPR_1, like in Chapter 3, Section 3.4 (if needed with its extension from Chapter 4 Section 4.4), but with the constraints that $\overline{c_1^1} \in [\min(c_1^1, c_1^2), \max(c_1^1, c_1^2)]$, $\overline{c_2^1} \in [\min(c_2^1, c_2^2), \max(c_2^1, c_2^2)]$, and $\overline{c_3^1} = c_3^1 = c_3^2$. Because we want to estimate the worst performance, the number of misses is maximized instead of minimized.

Let us assume that, for example, for the input data in_1 the application minimum number of misses is denoted by M_1 and it is given by the following:

$$M_1 = miss_1(c_1^1, in_1) + miss_2(c_2^1, in_1) + miss_3(c_3^1, in_1). \tag{5.3}$$

where $miss_{1,2,3}$ are the number of misses experienced by the three tasks of the application, when processing data in_1. Thus for input in_1 and any valid partition OPR the largest number of misses is given by the following:

$$\overline{M_1} = miss_1(\overline{c_1^1}, in_1) + miss_2(\overline{c_2^1}, in_1) + miss_3(\overline{c_3^1}, in_1). \tag{5.4}$$

The same type of investigation can be done for in_2 also and the values $\frac{M_1}{\overline{M_1}}$ and $\frac{M_2}{\overline{M_2}}$ reflect the robustness of the system to input data.

In media applications, time deadlines are imposed for processing a number of data units, for example a video decoder might have to decode 25 frames in a second. Therefore, it is also interesting to evaluate the variations in L2 behavior caused by different data units belonging to the same input stream. This means that, for instance, input data in_1 may be the first frame of a video stream and in_2 may be the next frame of the same video stream. Such a stability evaluation is useful because it gives a bound of the dynamic behavior that the application exhibits as a reaction of the input stream variations.

For a general application having N tasks $\mathcal{T} = \{T_i\}_{(i=1,N)}$ and M common regions $\mathcal{CR} = \{CR_j\}_{j=1,M}$, let $IN = \{in_l\}_{(l=1,L)}$ be the set of relevant input data sets. To express the allocated cache size c, we use the same index i to refer to tasks as well as to common regions. For the sake of simplicity we can consider that the first N values of c_i correspond to the application tasks and the next M (from $N+1$ to $N+M$) correspond to the application common regions. A task T_i's or a common region CR_i's number of misses $miss_i(c_i, in_l)$ depends on task's allocated L2 size c_i and on the input data in_l. When the application processes the input data in_l, its number of misses, is denoted with M_l and it is given by the following:

$$M_l = \sum_{i=1}^{N} miss_i(c_i^l, in_l). \tag{5.5}$$

For every input data $in_l \in IN$ the best partitioning ratio OPR_l is the set of tasks' allocated cache sizes $\{c_1^l, c_2^l, ..., c_N^l\}$. As previously mentioned, it is possible that the best partitioning ratio OPR_l differ among each other. The final partitioning ratio, $OPR = \{c_1, c_2, ..., c_N\}$ can be OPR_1, OPR_2, ... , OPR_L or any compromise among them, that respects the following condition:

$$c_i \in \left[\min_{IN}\{c_i^l\}, \max_{IN}\{c_i^l\}\right]. \tag{5.6}$$

and has the total cache allocated to tasks smaller than the available cache size C, $\sum_{i=1}^{N} c_i \leq C$.

In order to estimate an upper bound of the potential performance degradation in the case of in_l we calculate the worst partitioning ratio that respects the previous condition. We denote this ratio as being $\overline{OPR_l} = (\overline{c_1^l}, \overline{c_2^l}, ..., \overline{c_N^l})$. To determine $\overline{OPR_l}$ we use the same calculation method as for OPR_l, with the constraints that $\overline{c_i^l} \in \left[\min_{IN}\{c_i^l\}, \max_{IN}\{c_i^l\}\right]$ and instead of minimizing the number of misses, we maximize it (we are looking for the worst behavior). The application largest number of L2 misses under the previous conditions is denoted with $\overline{M_l}$, and it is given by the following formula:

$$\overline{M_l} = \sum_{i=1}^{N} miss_i(\overline{c_i^l}, in_l). \tag{5.7}$$

We define the application's stability $stab_l$ to in_l as being the relative variation between M_l and $\overline{M_l}$:

$$stab_l = \frac{M_l}{\overline{M_l}} \times 100\%. \tag{5.8}$$

The overall application stability is defined as the worst stability over the set of input data *IN*:

$$stab_A = \min_{IN}\{stab_l\}.$$

If the stability is close to *100*% the application behaves good for all its representative input data, so it is externally robust. If the difference between $\overline{M_l}$ and M_l is large, the static cache partitioning is not robust to input data variations and for better performance a dynamic repartitioning should be considered.

5.3 Experimental results

5.3.1 Internal Robustness

As aforementioned, we experiment on two types of applications, some consisting of communicating tasks and some consisting of independent tasks. The applications consisting of communicating tasks are described in YAPI, thus the data exchange and synchronization among the tasks is done through blocking FIFOs. A task is blocked (and consequently its processor switches to other task) when it has no available input data or output buffer space. On our experimental platform, for the purpose of the investigations, we induce higher task switching rate by shrinking the FIFOs sizes. For FIFOs larger than a certain size the task switching rate does not decrease anymore because a value intrinsic to the application is reached. We consider this lowest value as the reference task switching rate, as defined in the Section 5.1. In our case, both the PiPTV and the H.264 applications have the least number of misses for the lowest task switching rate. The internal robustness is relative to this number of misses, therefore the presented results reflect the largest deviations.

For the communicating tasks, the investigated average task switching rate values start at 41K and 24K times per second, corresponding to 4KB FIFOs and 2KB FIFOs for the H.264 and PiPTV, respectively. The task switching rate range ends at 74K and 41K times per second, corresponding to 0.5KB FIFOs and 0.4KB FIFOs for the H.264 and PiPTV, respectively. For FIFOs larger than 4KB and 2KB, for the H.264 and PiPTV respectively, the average task switching rate does not decrease anymore because the value intrinsic to the application is reached. For FIFOs smaller than 0.5KB for H.264 and 0.4KB for PiPTV, the applications deadlock, so the average task switching rate cannot be increased anymore. The measurements indicate that these task switching variation account for 30% and 66% difference in the number of L2 accesses for the H.264 and PiPTV, respectively.

In the case of applications composed from independent tasks, the task switching rate depends on the task scheduler policy. We enforced a policy that

preempts tasks with a rate ranging from 40K times per second to 400 times per second. This range is chosen to cover a large variety of possibilities. We consider the reference task switching rate as being the lowest one, therefore the internal robustness is relative to the number of misses encountered on that case.

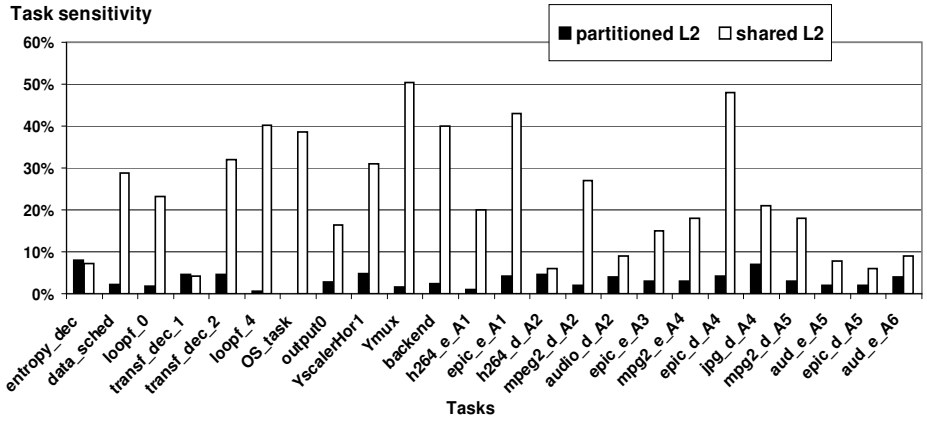

Figure 5.2: Tasks sensitivity: shared vs. partitioned cache.

For both application types, the L2 sensitivity of tasks is compared for the partitioned and the shared cache case (Figure 5.2). Due to space reasons (the sum of the number of tasks of PiPTV and H.264 is 69), in Figure 5.2 are depicted only the tasks that have the sensitivity larger than 2% in the partitioned

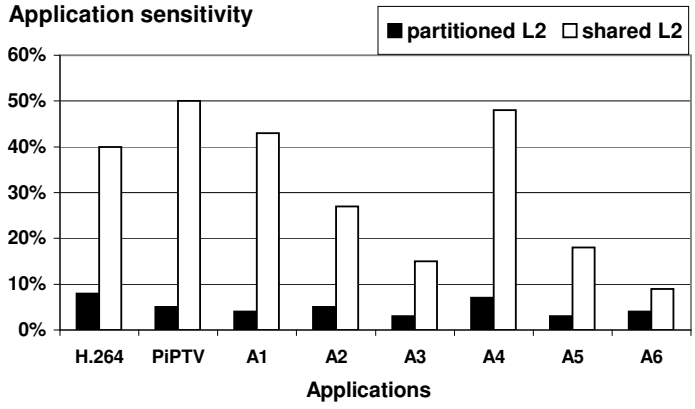

Figure 5.3: Application sensitivity: shared vs. partitioned cache.

application	H.264	PiPTV	A_1	A_2	A_3	A_4	A_5	A_6
max L2 variation	2%	7%	16%	9%	16%	12%	20%	12%

Table 5.1: Maximum variation in the L2 size allocated to a task.

cache case or larger than 20% in the shared cache case. In this figure it can be observed that, in general, the shared L2 is more sensitive than the partitioned one or their sensitivities are pretty close. Among the tasks that are not depicted in Figure 5.2, there are few ones for which the sensitivity of the partitioned L2 cache is larger than the one of the shared cache. However, for those few tasks, the sensitivity is smaller than 0.5%, so they do not influence the general observed trend, i.e., the shared cache is more sensitive that the partitioned one. Figure 5.3 presents the application sensitivity for all the eight applications. Over all the applications, the shared cache is on average 6 times more sensitive to task switching than the partitioned one. The largest sensitivity was observed at the applications H.264, PiPTV and A_4 for the case of partitioned and shared cache, respectively. For a partitioned cache, over the investigated task switching range, the application sensitivity as defined in Section 5.1 is at most 8%, with an average of 4%. For a shared cache, over the investigated task switching range, the application sensitivity as defined in Section 5.1 is at most 50%, with an average of 33%. These results reinforce the conclusions of Chapters 3 and 4 that suggest that, for the analyzed applications, partitioning the L2 is enough to achieve compositionality to a large extent.

5.3.2 External Robustness

As detailed in Section 5.2, the best cache partitioning ratio of an application varies with its tasks input data. In order to quantify the differences among the best cache partitioning ratios, we use the maximum variation of the L2 size allocated to a task, across different input streams. We calculate the partitioning ratios for each task, for each input stream, and we present in Table 5.1 the maximum variation in the allocated L2 size over all tasks, per application. The values in Table 5.1 are relative to the total cache size available to each application. In general we found that the differences among the best partitioned ratio corresponding to different input data are relatively small. As one can observe in Table 5.1, over the 8 applications that we exercised, the cache of a task varies at maximum with 20% from the total cache size.

For some input data, the partitioning ratio is non-optimal and this induces a performance degradation. To quantify this degradation, in Section 5.2 we

input data	in_1	in_2	in_3	in_4
H.264	96%	96%	100%	98%
PiPTV	92%	100%	93%	98%
A_1	100%	93%	93%	96%
A_2	90%	100%	91%	97%
A_3	97%	93%	100%	90%
A_4	95%	91%	100%	95%
A_5	100%	95%	98%	96%
A_6	92%	91%	93%	100%

Table 5.2: Application stability for different input data.

introduced the stability metric. In Table 5.3.2 the stabilities corresponding to each application are illustrated. For all the eight applications we investigated four different input data streams. For the six application consisting of non-communicating tasks, each task needs its own input data. In all cases, we would like to mention that the set of input data corresponding to an application has the same size and the same "quality" level for each experiment. In this section we do not investigate the effects of things like enlarging the resolution or the scaling factor of a video stream, or changing the encoding quality of an image. The reason behind is that an application that produces different resolution/quality/etc. output, has actually some different tasks, and the quality change can be regarded as a scenario change, case tackled by the next chapter.

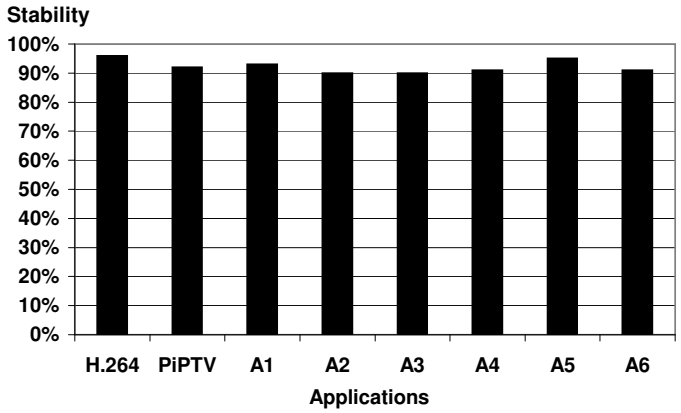

Figure 5.4: Minimum application stability.

Figure 5.4 presents, for each of the eight applications, the minimum stabil-

ity over the set of four input streams. We observe that the minimum stability of each application is pretty high, ranging over the eight applications from 90% to 96%, with an average of 92%. Taking these facts into account, we can conclude that all the eight applications are quite robust to input stimuli in the presence of static cache partitioning. A stability comparison between the shared and the partitioned cache is not possible because the stability, as defined in Section 5.2, is linked to the partitioned ratio, thus it cannot be computed for the shared cache scenario.

5.4 Conclusion

In this chapter we proposed a method to analyze the static cache partitioning robustness. We define and discuss two types of robustness: internal (determined by inter-task interference in the L1 cache) and external (determined by the variations of the L2 behavior due to various input data sets). For both types of robustness we introduced quantification metrics. For internal robustness we defined the sensitivity function, which measures the variation of L2 misses caused by the L1 variations over a range of task switching rates. For external robustness we defined the stability function, which measures the performance deviation for the case the application processes another input data set than the one utilized to determine the static L2 partitioning ratio.

To demonstrate our approach we analyzed both types of parallel applications introduced in Chapter 2: (1) applications consisting of communicating tasks and (2) applications consisting of independent tasks. Concerning the internal robustness, if the cache is partitioned, the application sensitivity is at most 8%, with an average of 4%. This small sensitivity reinforces the conclusion that partitioning the L2 is enough to achieve compositionality in a large degree, for these applications. Comparing the internal robustness of the shared and partitioned cache cases, we found that the shared cache is on average 6 times more sensitive than the partitioned one. Moreover, the large difference among the shared cache and the partitioned cache sensitivity is an interesting fact on itself. It suggests that the optimizations processes for L1 and L2 caches can be decoupled if the L2 is managed on a task centric manner. Concerning the external robustness, the variations induced in the L2 behavior by various input data sets are at most 10% over all the applications we experimented. This accounts for an average stability of 92%, therefore, for the investigated applications, we can conclude that the static cache partitioning is quite robust with respect to input stimuli variations.

Chapter 6

Dynamic task centric cache management

In the previous chapters we presented a task centric approach for static cache management, in order to ensure compositionality for media applications. In this chapter we extend this static cache partitioning method with a dynamic task centric cache management strategy. For example, a multimedia application may have multiple execution scenario, in the sense that some tasks may start and/or stop. Let us consider the example of a mobile device with video and sound facilities. The video decoding task should be active only when a video stream is on display, and not when the user just listens to music, for instance. If the cache is statically allocated, the part corresponding to the video decoder would be reserved all the time, thus also when the task is stopped. In this manner the cache resource is waisted and the system performance may be penalized.

In this chapter we propose a strategy to dynamically repartition the cache at a scenario change, such that the compositionality is enhanced and the entire cache is efficiently utilized [73], [68]. This strategy is based on determining the best static partition for each scenario, and dynamically changing the partitions on a scenario switch. In order to keep data correctness, our cache repartitioning implies flushing, therefore a time penalty. This is especially critical for task that have a low tolerance to perturbations. To cope with this problem we first propose a design-time method, to determine each task's cache footprint in each scenario, such that (1) in particular the critical tasks are protected against cache perturbation, and (2) in general the number of necessary flushes are minimized. Furthermore, we propose a partial cache flush policy that ensures that the statically calculated footprints are respected and further

decreases the penalty by flushing only what it is necessary, as late as possible, in the eventuality the data flush is actually not needed anymore.

The overview of this chapter is as follows. Section 6.1 introduces the cache performance problems encountered at set-based cache repartitioning. In Section 6.2 we present the off-line part of the cache repartitioning method. Section 6.3 presents the on-line part of the cache repartitioning. The experimental results are presented in Section 6.4 and finally Section 6.5 concludes the chapter.

6.1 Cache repartitioning

We start this section by first reminding some useful notations and then we detail the implications of set-based cache repartitioning. We consider that in each scenario S_q only a subset of tasks $\mathcal{T}_q \subseteq \mathcal{T}$ is active and that in a scenario S_q the cache size of a task $T_i \in \mathcal{T}_q$ is denoted with $c_{i,q}$. Naturally, if a task T_i is stopped in a scenario S_q, then $c_{i,q} = 0$. The allocable cache units of an L2 cache are numbered from *1* to *C*. We define the cache footprint of a task T_i ($T_i \in \mathcal{T}_q$) as the contiguous cache interval where T_i data reside, $cf_{i,q} = [b_{i,q}, b_{i,q} + c_{i,q})$, where the $b_{i,q} \in [1, C - c_{i,q}]$ represents the cache unit where T_i's footprint begins. The cache footprint of an entire application in the scenario S_q, is the collection of each task cache footprints $\{cf_{i,q}\}$, with $T_i \in \mathcal{T}_q$. Moreover we consider that some tasks are more sensitive to scenario switching perturbations than others, in the sense that the output quality severely degrades if such a sensitive task is even lightly disturbed. We denote such tasks as critical. The critical tasks definition is the job of the application designer and it does not represent the subject of this thesis.

In Chapter 3 we present an in depth quantitative comparison among the static set and associative cache partitioning. The conclusion of this comparison, for the class of applications that we are interested on, is that the associativity based partitioning achieves compositionality, but severely degrades the cache performance. As our method dynamically switches among static partitions, with a coarse time quantum (we primarily target scenario switching frequencies above 10Hz) it is legitimate to extrapolate that the trend observed for static partitioning holds true also for the dynamic partitioning. Thus, in this chapter we build our dynamic cache management method starting from a set-based partitioned cache.

Now, let us take a closer look at what is happening when the cache is repartitioned. We consider a transition from the current scenario S_q to the new scenario S_w ($S_q \rightarrow S_w$). We remind that, due to implementation reasons the

number of sets a task can get allocated is a power of two. We consider that the data are mapped to cache sets using a conventional *modulo* function [38]. Thus an address X accessed by task T_i is cached on the current scenario S_q in the set $set_{X,q} = b_{i,q} + X\%c_{i,q}$. On the next, new, scenario S_w, the same address is cached on $set_{X,w} = b_{i,w} + X\%c_{i,w}$. In such situations the cache location of X might change at the $S_q \rightarrow S_w$ transition, therefore precautions should be taken such that later on T_i's sees the most recent value at X. If $cf_{i,q} \cap cf_{i,w} = \{\varnothing\}$, after the scenario transition eventually all T_i data have to move into the new T_i's cache part. We would like to mention that in this thesis we do not assume the existence of a possibility to directly transfer data from one L2 set to the other, nor the existence of a mechanism (similar to a cache coherence protocol) that can look in multiple L2 sets to determine where is the most recent data value requested. For now, our option is to flush the T_i's footprint corresponding to the old scenario S_q. Later on when a data item is needed it is loaded from the main memory. This strategy implicitly moves a data item from one cache set to the other, via the main memory.

In the following we investigate the cases when the cache content can be reused, at scenario change. For simplicity sake, let us assume that for a scenario switch both cache footprints begin on the same cache set ($b_{i,q} = b_{i,w}$) and the cache sizes vary with a factor of *2*. Thus there are two possibilities at a scenario change:

(1) The cache doubles ($cf_{i,q} \subset cf_{i,w}$, $c_{i,w} = 2 \times c_{i,q}$). This example is illustrated in Figure 6.1. Let us assume that in S_q an address X maps in the cache in $set_X = b_{i,q} + X\%c_{i,q}$. Moreover, for the same scenario S_q, the data at address $X + c_{i,q}$ also maps in set_X. When the cache doubles at $S_q \rightarrow S_w$, the data at address X still maps in set_X, but the data at address $X + c_{i,q}$ maps in $set_X + c_{i,q}$. As one can see, not all data in the $cf_{i,q}$ cache footprint stay in the same location in the double sized footprint $cf_{i,w}$. Therefore, to keep data correctness, one has to flush only the T_i's data that does not map anymore in $cf_{i,w}$ in S_w. As already mentioned, we do not assume the existence of a mechanism to keep such cache lines coherent between scenarios, thus for the present work the entire $cf_{i,q}$ is flushed.

(2) The cache halves ($cf_{i,w} \subset cf_{i,q}$, $c_{i,q} = 2 \times c_{i,w}$). As visible in Figure 6.2, each data item present in S_q in the first $c_{i,w}$ sets of $cf_{i,q}$ is mapped in the same place in the scenario S_w. For those data items $X\%c_{i,q} = X\%c_{i,w}$, because $c_{i,q} = 2 \times c_{i,w}$. However, the other data for which in S_q $X\%c_{i,q} > c_{i,w}$ (for instance $X + c_{i,w}$, as illustrated in Figure 6.2) are relocated in $cf_{i,w}$, when the scenario becomes S_w. In conclusion, in order to keep data correctness, only

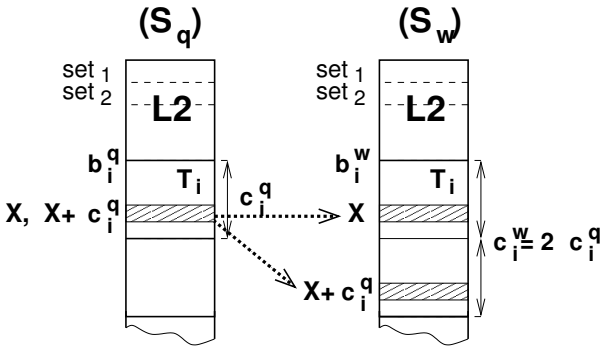

Figure 6.1: Cache repartitioning - doubling the size

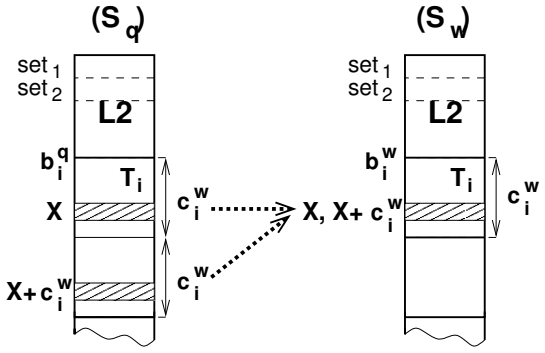

Figure 6.2: Cache repartitioning - halving the size

the second half of $cf_{i,q}$ has to be flushed.

A similar rationale applies when a task's cache size increases or decreases with more than a factor of *2* between consecutive scenarios. In conclusion, on an $S_q \rightarrow S_w$ transition, there are two cases when the cache content of a task T_i can be reused: (1) if T_i's cache footprint stays the same, and (2) if T_i's number of cache sets decreases, and if the starting set of the new cache footprint $b_{i,q} = b_{i,w} + \varkappa \cdot c_{i,w}$ with $\varkappa \in \mathbb{N}$, $\varkappa < \frac{c_{i,q}}{c_{i,w}}$.

As mentioned, unrelated to the compositionality, cache partitioning offers a optimization freedom. In Chapter 3, we identify two optimization problems as being of interest: (1) the *Cache Allocation Problem* $\mathcal{CAP}$ (finding c_i), and (2) the *Cache Mapping Problem* $\mathcal{CMP}$ (finding b_i). As we could see, for static cache partitioning only $\mathcal{CAP}$ is of interest, the footprints being unrelated to the cache performance. However, for dynamic cache partitioning this does

not hold true anymore.

In the case of set based repartitioning, the repartitioning costs may depend on $b_{i,q}$ and $b_{i,w}$. For a good understanding of this phenomenon, we present in the following a simple example, involving an application *A* with four tasks: T_1, T_2, T_3, and T_4. We consider three scenarios, S_1, S_2, and S_3. For these scenarios, the cache footprints are illustrated in Figure 6.3. Let us assume that the application switches through the three scenarios in order, $S_1 \rightarrow S_2 \rightarrow S_3$. In this case the following happens. At $S_1 \rightarrow S_2$ the cache of T_2 is flushed as its size increases. The T_3 cache is not flushed as T_3 stays in the same place in cache in both scenarios. At $S_2 \rightarrow S_3$ the cache of T_2 is flushed, as T_2 is relocated in the cache. T_1 restarts, and so the flushing its previous footprint is considered now (more details about the flushing strategy will follow in Section 6.3). However, in this example T_1 has the same footprint in S_1 and S_3, therefore no flushing is required for its cache part. Moreover, T_3's cache shrinks, but as this task still owns set_4, only set_3 has to be flushed.

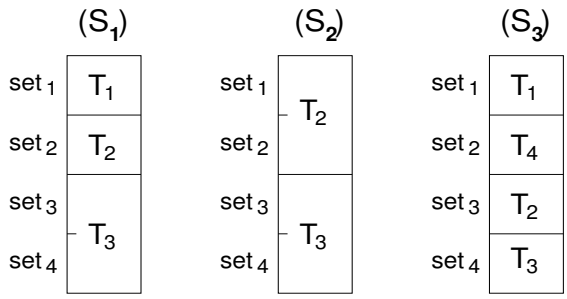

Figure 6.3: Cache flush example

The amount of cache flushing depends on the scenario order and on the lines where a task cache footprint begins. For instance in the previous example, in the scenario S_3, the tasks footprints in cache might have had the order T_1, T_2, T_4, T_3 ($b_1 = 1$, $b_2 = 3$, $b_3 = 4$, $b_4 = 2$) instead of T_1, T_4, T_2, T_3 ($b_1 = 1$, $b_2 = 2$, $b_3 = 4$, $b_4 = 3$), i.e. the positions of T_4 and T_2 might have been inverted. In this case only a partial flush of set_1 would have been necessary to ensure T_2 correctness and T_2 would have reused half of its footprint. Moreover, if for instance T_2 is critical, its execution is unacceptably interrupted by cache flushes. In conclusion, in dynamic repartitioning the performance of the systems relies heavily on $b_{i,q}$ and $b_{i,w}$, therefore the *cache mapping problem* becomes interesting.

In the next sections we first discuss and solve the cache mapping problem,

and then we present the run-time cache management strategy.

6.2 Cache content reuse via footprint management

In this section we consider that the cache allocation problem is solved for each application scenario. For this purpose one can use the methods proposed in Chapters 3 and 4 that minimize the total application number of misses or the throughput. Moreover, we assume that the application designer takes care that the allocated cache sizes of the critical tasks are constrained to have the same size in each scenario. We define the CCR_i of a task T_i as being its cache content reuse, during an execution:

$$CCR_i = \sum_{\substack{S_q \to S_w \\ c_{i,q} > c_{i,w} \\ b_{i,q} = \nvDash b_{i,w}}} c_{i,w} \cdot p_{q \to w} \tag{6.1}$$

where $p_{q \to w}$ is the probability that S_w immediately succeeds S_q in that execution.

The cache content reuse of an entire application CCR is the sum of the cache reuse exhibited by its tasks:

$$CCR = \sum_{i=1}^{N} CCR_i. \tag{6.2}$$

The cache mapping problem that we investigate in this section is formulated as follows. Given: (1) an application A consisting of a task set $\mathcal{T}$ and having $\mathcal{S}$ scenarios, (2) the transition probability (or the relative frequency) among each scenario pair $p_{q \to w}$, and (3) the tasks cache sizes in each scenario $c_{i,q}$, the objective is to find the beginning cache line $b_{i,q}$ of each task footprint in each scenario ($cf_{i,q} \cap cf_{j,q} = \{\varnothing\}, \forall T_i \in \mathcal{T}_q, \forall T_j \in \mathcal{T}_q, i \neq j$) such that the cache content reuse is: (1) complete for the critical tasks and (2) maximized for the other tasks.

A "complete reuse" is achieved when a task has the same cache footprint in all the consecutive scenarios in which it is active. Critical tasks may also stop, as some utilization scenarios do not require their execution. The important thing is that a critical task should not be disrupted as long as it is active. When it becomes inactive, its cache may be used by other tasks.

In the next subsections we first prove the NP-completeness of the cache mapping problem, then we formulate it as an Mixed Integer Linear Problem for an optimal solution, and after that we propose a fast heuristic to solve this problem.

6.2.1 Hardness of the cache mapping problem

In this subsection we prove that the cache mapping problem is equivalent with the Dynamic Storage Allocation Problem $\mathcal{DSAP}$ (addressed as SR2 in [33]), that is known to be NP-hard.

In order to prove this we first recall the dynamic storage allocation problem, with the notations from [33]. Note that these notations may clash with the ones used for $\mathcal{CMP}$. To avoid confusion, we explicitly specify to which problem we refer. In $\mathcal{DSAP}$ given are: (1) a set of items to be stored $a \in A$ of size $s(a) \in \mathbb{Z}^+$, arrival time $r(a) \in \mathbb{Z}_0^+$, and departure time $d(a) \in \mathbb{Z}^+$ and (2) a storage size $D \in \mathbb{Z}^+$. The question is if there exists a feasible storage allocation $\sigma : A \rightarrow \{1, 2, ...,D\}$ such that for every $a \in A$ the allocated storage interval $I(a) = [\sigma(a), \sigma(a) + s(a)-1]$ is contained in $[1,D]$ and such that, for all $a, a' \in A$, if $I(a) \cap I(a') \neq \{\varnothing\}$, then either $d(a) \leq r(a')$ or $d(a') \leq r(a)$.

To prove the equivalence of the two problems ($\mathcal{CMP}$ and $\mathcal{DSAP}$) we make the following notations and $\mathcal{CMP}$ reductions:

1. for simplicity reasons we can assume that a scenario takes one time unit, as no task may start or stop during a scenario (hence the cache allocation events occur between scenarios),

2. we restrict the generality of the $\mathcal{CMP}$ by considering that a task T_i has the same cache size c_i in each scenario in which it is active ($c_{i,q} = c_{i,w} = c_i, \forall S_q, S_w, T_i \in \mathcal{T}_q, T_i \in \mathcal{T}_w$), as if all the tasks are critical,

3. we consider a given scenario sequence of length U, $\{s_k\}_{(k=1,2,..,U)}, s_k \in \mathcal{S}$, therefore all the $p_{q \rightarrow w}$ are known, and

4. for each task T_i we address the longest subsequence of consecutive scenarios set in which T_i is active with $\{ss_{i,m}\}_{(m=1,2,...,U_i)}$, $\{ss_{i,m}\} \subset \{s_k\}$. There is no reasons to assume that some of T_i's data might still be in the cache when T_i is restarted, after a time it was inactive (in the scenarios between two consecutive subsequences, $ss_{i,m}$ and $ss_{i,m+1}$ other tasks execute, possibly using T_i's cache). Thus, from the cache's point of view, it is like T_i is replaced by U_i tasks, each of them having the same

functionality as T_i and a c_i cache size, but the U_1 executes only in all scenarios from subsequence $ss_{i,1}$, the U_2 executes only in all scenarios from subsequence $ss_{i,2}$, etc. Consequently, we replicate each task T_i of the application in U_i tasks. As a result *A* is described by a set $\mathcal{T}'$ of *N'* tasks and each $T'_i \in \mathcal{T}'$ has, by construction, only one scenario subsequence in which it is active. For this case we define the arrival scenario $r(T'_i)$ and the departure scenario $d(T'_i)$ as the first and the last scenario in which T'_i is active. Moreover we denote with b'_i the cache units where T'_i footprint begins its active scenario subsequence.

Let us present a simple example with a sequence of 4 scenarios, to give an intuitive idea about this task replication. May T_i be active in both S_1 and S_2 then inactive in S_3 and later back active in S_4. For maximum reuse, T_i has the same footprint in S_1 and S_2 (if T_i is critical, it must have the same footprint in both scenarios). However, in S_3 T_i is stopped, therefore another task may use T_i former cache (this is possible regardless of whether T_i is critical or not, as we consider that critical tasks should not be disrupted as long as they are active and may be flushed out of the cache when they are inactive). Later on, in S_4 the task T_i is active again, but its cache may be flushed during S_3, therefore the situation is like T_i restarted with a cold cache. In this case, the cache behaves like we would have two tasks T'_i and T''_i (with $c'_i = c''_i = c_i$), the first one being active in S_1 and S_2 and inactive in the rest of the scenarios, and the second one being active only in S_4.

With these simplifications, the cache mapping problem can directly transform into the dynamic storage allocation problem, because the following relate to each other in a one-to-one fashion (first we mention the $\mathcal{CMP}$ variables and then the $\mathcal{DSAP}$ ones):

1. the tasks T'_i and the items *a*;
2. the cache size c'_i and the item size *s(a)*
3. the total cache size *C* and the storage size *D*;
4. the arrival and departure scenarios *r(T'_i)* and *d(T'_i)* of T'_i and the arrival and departure time of *a r(a)* and *d(a)*, respectively;
5. the function of the cache units where a footprint begins b'_i and the feasible storage function σ;
6. the T_i cache footprint and the allocated storage interval *I(a)*;

7. the fact that two tasks may share a cache part only when they are not active in the same time and the condition that two items *a, a'* $\in A$, if *I(a)* $\cap$ *I(a')* $\neq \{\varnothing\}$, then either *d(a)* $\leq$ *r(a')* or *d(a')* $\leq$ *r(a)*.

Taking into account these presented facts, we can conclude that $\mathcal{CMP}$ is equivalent with $\mathcal{DSAP}$, and therefore that $\mathcal{CMP}$ is NP-hard.

6.2.2 Optimal solution for the cache mapping problem

In this section we present a Mixed Integer Linear Problem (MILP) formulation for the cache mapping problem. We use a set of *0/1* variables $\{l_{i,q}^k\}$, *(i=1,2,...,N, q=1,2,...,U, k=1,2,...,C)* to indicate if the cache footprint of a task T_i in scenario S_q starts at the cache line k ($l_{i,q}^k = 1$) or not ($l_{i,q}^k = 0$). Naturally, a task footprint may start only at one cache line, therefore for each $1 \leq i \leq N$ and $1 \leq q \leq U, T_i \in \mathcal{T}_q$:

$$\sum_{k=1}^{C} l_{i,q}^k = 1. \tag{6.3}$$

Within a scenario, a cache line k may not be allocated to more than one task. A case for which this is simple to express, is the one of the first cache line *(k=1)*, where a single footprint may begin:

$$\sum_{i=1}^{N} l_{i,q}^1 \leq 1. \tag{6.4}$$

The second line *(k=2)* may be the start of a footprint or, it may belong to the footprint started in line *1*, if this last one has a size larger than *2*. This can be express in MILP as follows:

$$\sum_{i=1}^{N} l_{i,q}^2 + \sum_{\substack{i=1 \\ c_{i,q} \geq 2}}^{N} l_{i,q}^1 \leq 1. \tag{6.5}$$

These constraints are generalized for an arbitrary cache line k as follows:

$$\sum_{i=1}^{N} \sum_{\substack{v=1 \\ c_{i,q} \geq v}}^{k} l_{i,q}^{k-v+1} \leq 1. \tag{6.6}$$

The cache line where a footprint starts is given by the formula:

$$b_{i,q} = \sum_{k=1}^{C} k \cdot l_{i,q}^{k}. \tag{6.7}$$

In order to indicate the situations in which the cache of T_i is reused in scenarios S_q and S_w, namely: (1) $b_{i,q} = b_{i,w}$, or (2) $b_{i,q} = b_{i,w} + \varkappa \cdot c_{i,w}$, with $1 < \varkappa < \frac{c_{i,q}}{c_{i,w}}$, $\varkappa \in \mathbb{N}$ we introduce a *0/1* variable, $y_{q,w}^{i}$. In this case Equation (6.2) of the cache content reuse for an application becomes:

$$CCR = \sum_{i=1}^{N} \sum_{\substack{S_q \to S_w \\ c_{i,q} \geq c_{i,w}}} y_{q,w}^{i} \cdot c_{i,w} \cdot p_{q \to w} \tag{6.8}$$

In the following we present the MILP definition for the *0/1* variable $y_{q,w}^{i}$, that specifies whether $b_{i,q} = b_{i,w}$. In the case $c_{i,q} > c_{i,w}$, a similar equality have to be checked for each $\varkappa$ ($1 < \varkappa < \frac{c_{i,q}}{c_{i,w}}$, $\varkappa \in \mathbb{N}$) for which $b_{i,q} = b_{i,w} + \varkappa \cdot c_{i,w}$, but in principle the formulation is no different.

To give a formula for $y_{q,w}^{i}$, we use two intermediary *0/1* variables $g_{q,w}^{i}$ and $h_{q,w}^{i}$ that express if $b_{i,q} \leq b_{i,w}$ and $b_{i,q} \geq b_{i,w}$, respectively. If both are *1* in the same time, then $b_{i,q} = b_{i,w}$, thus $y_{q,w}^{i}$ should be *1*, otherwise it should be *0*. For clarity reasons, in the following we omit the *i, q, w* indexes of the *y, g, h* variables. The next two equations represent the constraints for g.

$$C \cdot g \leq b_{i,q} - b_{i,w} + C; \tag{6.9}$$

$$(C+1) \cdot g \geq b_{i,q} - b_{i,w} + 1; \tag{6.10}$$

As h expresses the same type of relation among two integers (namely inequality), the constraints for h are similar than the ones for g, except that in place of $b_{i,q}$ is $b_{i,w}$ and vice versa. Having defined g and h, y is given by the following two equations:

$$y \geq g + h - 1. \tag{6.11}$$

$$2 \cdot y \leq g + h \tag{6.12}$$

In summary, in this subsection we introduced a Mixed Integer Linear Problem formulation that solves the cache mapping problem such that the amount

of cache reuse is optimized. As one can notice this formulation involves as many as *(C+3 × U+1) × N × U* variables. For a concrete case of 4 tasks, 7 scenarios and a 512 KB L2 the number of variables required reaches 7784. It is a known fact that the time needed by an MILP solver to provide a solution is in direct relation with the number of variables in the formulation [23]. As one can see, the MILP formulation of $\mathcal{CMP}$ easily becomes too large to be solved in reasonable time, hence in the next subsection we propose a heuristic.

6.2.3 Heuristic for the cache mapping problem

In the heuristic approach to $\mathcal{CMP}$, as a first step, the cache mapping problem for the entire application is split into several smaller instances of the same problem.

If a task subset $\Psi \subset \mathcal{T}$ has its cache size sum constant over all scenarios $\left(\sum_{q=1}^{U} \sum_{T_i \in \Psi} c_{i,q} = \Gamma\right)$, then Ψ and $\mathcal{T} \setminus \Psi$ are two disjoint task subsets that behave as if each one of them is an independent application having the cache size Γ, and $C - \Gamma$, respectively. In this manner the problem can be further recursively split, obtaining a set of task subsets $\{\Psi_m\}_{(m=1,2,...,U)}$, $\bigcup_{m=1}^{U} \Psi_m = \mathcal{T}$, $\Psi_m \cap \Psi_k = \{\varnothing\}, m \neq k$. In order to build the $\{\Psi_m\}$ subsets we have to generate all possible tasks subsets and test if they respect the condition that the sum of their cache sizes is constant over all scenarios. Thus the number of iterations that are executed is $C_N^1 + C_N^2 + ...C_N^{\frac{[N+1]}{2}}$, where $C_N^k = \frac{N!}{k!\cdot(N-k)!}$. Even though the complexity of building the $\{\Psi_m\}$ subsets is not polynomial, this does not constitute a problem in practice, as the number of tasks is in the order of *O(10)*.

In this paragraph we give an example meant to illustrate the separation of tasks $\mathcal{T}$ into subsets $\{\Psi_m\}$ and to highlight the mechanisms behind the $\mathcal{CMP}$ heuristic. This example uses 4 tasks and 3 scenarios, with the following characteristics: T_1 is active all the time and has the same cache size in all 3 scenarios, T_2 is active only in S_1 and S_2, and has different cache sizes in the two scenarios, and T_3 is active in S_1 and S_3 and has different cache sizes in the 2 scenarios and T_4 is active in S_2 and S_3. Figure 6.4 presents the cache of the 4 tasks in the 3 scenarios, for a possible cache map. As also visible in Figure 6.4, the 4 tasks can be separated in two subsets, such that the first subset Ψ_1 contains only the task T_1 (that has the same cache size in all scenarios), and the second subset Ψ_2 contains $\{T_2, T_3, T_4\}$ (the sum of T_2, T_3 and T_4 cache

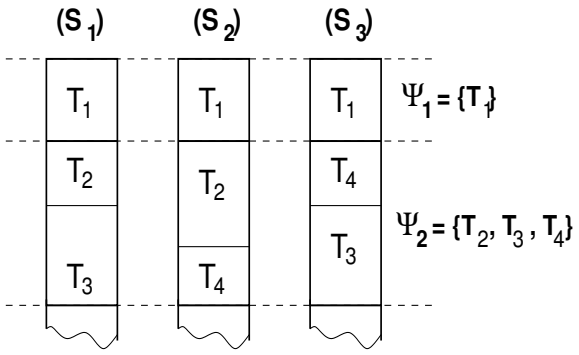

Figure 6.4: Example: L2 cache footprints

parts are always the same). As a result we now have two similar instances of the $\mathcal{CMP}$ problem: (1) find the cache map for $\Psi_1 = \{T_1\}$ when having $C = c_{1,q}, (\forall)q$=$\{1, 2, 3\}$ and, (2) find the cache map for $\Psi_2 = \{T_2, T_3, T_4\}$ when having $C = c_{2,q} + c_{3,q} + c_{4,q}, (\forall)q$=$\{1, 2, 3\}$. Solving $\mathcal{CMP}$ for Ψ_1 is straightforward, as Ψ_1 contains only one task. Note that splitting the tasks set in subsets guarantees that, for the subsets containing only one task have a complete cache reuse. In the case of Ψ_2 it can be observed that, for instance if in S_2 T_4 is placed in cache immediately after T_1, thus before T_2, at a scenario switch $S_1 \rightarrow S_2$ none of the T_2 data is reused, whereas maximum possible reuse of T_2 data is achieved when its place it is not changed. If T_2 is always the first task after at the top of the cache of Ψ_2), its reuse is maximum. Same observation is valid also for the reuse of T_3, that reaches its maximum if T_3 is always place at the bottom of Ψ_2. This fact represents the main idea of the $\mathcal{CMP}$ heuristic, as described in the remainder of this section.

For the general case in which Ψ_m contains N_m tasks, our mapping heuristic is described by Algorithm 2. As a general rule, the heuristic successively places task footprints in the cache in a decreasing order of their reuse CCR_i, starting from the extremities of the cache toward the middle, giving priority to critical tasks. At one mapping step we fix the footprint of a task T_i in each scenario in which T_i is active. This means that, if in scenario S_q a task T_i is mapped before a task T_j ($T_i, T_j \in \mathcal{T}_q$), also in a scenario S_w T_i is mapped before a task T_j ($T_i, T_j \in \mathcal{T}_w$). This strategy is based on the observation that the reuse tends to increase when the task have the same order in the cache in each scenario (see the example in the Figure 6.4). The reuse CCR_i is dependent on the task position in the cache and it is recalculated at each mapping step, taking in consideration the current values for $b_{i,q}$ and $b_{i,w}$. Given that a number of

tasks are already mapped in the cache, for the remaining tasks we define CCR_i^t and CCR_i^b as the the reuse if T_i is placed at the top (respectively at the bottom) of the free cache extremity. We denote $CCR_i^{tb} = CCR_i^t \cup CCR_i^b$. Furthermore, $\{T_m^{cr+ok}\} \subset \Psi_m$ is the subset of critical tasks with sane footprint if placed at the top or at the bottom of the free cache space.

Algorithm 2: Finding the cache footprint for all tasks

foreach $\Psi_m \in \{\Psi_m\}$ **do**
  while $\Psi_m \neq \{\varnothing\}$ **do**
    for $T_i \in \Psi_m$ **do** calculate CCR_i^{tb} and and form $\{T_m^{cr+ok}\}$;
    foreach $\{$*top, bottom*$\}$ *cache extremities* **do**
      if $\{T_m^{cr+ok}\} \neq \{\varnothing\}$ **then** place the $T_i \in \{T_m^{cr+ok}\}$ with the largest CCR_i^{tb};
      else place the $T_i \in \Psi_m$ with the largest CCR_i^{tb};
      $\Psi_m = \Psi_m \setminus T_i$;
    end
  end
end

If Algorithm 2 cannot sanely place all Ψ_m's critical tasks, we rerun it, but at step 5 and/or 6, instead of picking the task with the largest reuse we make it select the task with second, third, etc. largest reuse. In the case that after all possible backtracking in Ψ_m no sane solution is found, we merge Ψ_m with the Ψ_k subset that has the minimum number of critical tasks, and restart the entire optimization process. If no sane critical tasks placement is found even after merging all Ψ_m's, one of the following should be revised: (1) the cache sizes $c_{i,q}$ allocated to each tasks or (2) the total cache size or (3) the selection of the critical tasks. The first case actually means that the cache mapping influences cache allocation (i.e., they are performed simultaneously). This is an interesting problem by itself, and it can be subject for future research.

In practical situations the scenario transition frequency (or probability) may not be known at design-time. An extension that copes with run-time cache remapping, depending on the experienced $p_{q \to w}$ is possible. Anyway, the $\{\Psi_m\}$ set does not depend on the scenario switch frequency, therefore it can be already determined off-line. Then a CCR_i^{tb} formula with $p_{q \to w} = \frac{1}{U}$ (all transitions have equal probability) can be utilized to guide the initial footprint calculation. After that, at run-time, the system can learn the scenario transition frequencies, and adjust the footprints accordingly. If all the critical tasks can be placed on the first run of the Algorithm 2 the complexity of find-

ing the footprints is polynomial, thus it is suitable for run-time execution (this certainly holds true if, for example, every Ψ_m has at most two critical tasks). Nevertheless, a run-time solution independent of the number of critical tasks is another interesting follow up of the present work.

6.3 Run-time cache management

In order to control the cache repartitioning, we employ a software Run-Time Cache Manager (RTCM) executing on the control processor. At $S_q \rightarrow S_w$, the RTCM jobs are, in order: (1) to stop the tasks that are not active in the new scenario ($T_i \in \mathcal{T}_q, T_i \notin \mathcal{T}_w$) and the tasks that change their footprints ($T_i \in \mathcal{T}_q, T_i \in \mathcal{T}_w, cf_{i,q} \neq cf_{i,w}$); this strategy allows tasks that do not change their cache footprint to continue executing, reducing the flush impact, (2) to initiate a partial cache flush according to the reuse rules in Section 6.1, and to wait until the flush is performed, (3) to update the cache partitioning tables to the new cache footprint, and (4) to start the new tasks ($T_i \in \mathcal{T}_w, T_i \notin \mathcal{T}_q$) and to resume the tasks that changed their footprints ($T_i \in \mathcal{T}_q, T_i \in \mathcal{T}_w, cf_{i,q} \neq cf_{i,w}$). In addition, we propose a cache controller that provides partial flush, as introduced in the rest of this section.

In general, cache flushing implies a penalty that has two components. First it is the extra time required to write the content of the flushed lines in the main memory. Second, after the flush, extra (cold) misses may occur when the flushed data are needed again in the cache. To minimize these overheads we propose to flush only what it is necessary to ensure data correctness at each scenario change, and to delay the flush as long as possible, in the eventuality that it might not be needed anymore. The cache flushing policy consists of the following rules:

(1) *Flush no code*. On the CAKE platform the code cannot be modified during execution (it is read-only). Thus the main memory contains a valid copy of all the application instructions. As a results, correctness is preserved without having to flush the code.

(2) *Late flush*. This rule applies in the case a task T_i stops at a scenario change. Only at the moment when T_i resumes its execution, its data are flushed out of the cache (if, of course, T_i's cache location is changed). In the mean time some of the data might have been already swapped out by other tasks. In this manner some cold misses still occur, but a part of the flushing overhead is avoided. Moreover, if the task restarts and has the same cache part, it potentially benefits from some remaining cached data.

(3) *Flush only the valid, "owned", cache lines*. If the cache coherence mechanism marks a cache line as invalid, the memory hierarchy contains a more recent copy of the corresponding data, therefore the data correctness is not influenced by the content of that line. A cache line is considered as "owned" by a task T_i, if that line stores some of T_i data. Let us assume a scenario transition $S_q \rightarrow S_w$ when all T_i cache lines are relocated. In order to ensure the correctness of T_i's data, only the cache lines owned by T_i have to be flushed out of $cf_{i,q}$ (data belonging to another tasks may still be cached in some of $cf_{i,q}$ lines, from a previous execution, as allowed by the late flush strategy).

Besides the implementation of set based partitioning, the dynamic cache management requires that each cache line has a *task id*, in order to be able to check the line's ownership, as required by the third cache flushing rule. Moreover the lines caching code should be distinguished from the lines that cache data (in general L2s are unified), to support the first cache flushing rule. However, the storage involved in these two issues (*task id* plus 1 bit for code/data) is minor when compared to the total cache size (under 1% for an L2 having 512 Bytes cache lines).

6.4 Experimental results

In this section we investigate two issues related to cache repartitioning: the compositionality and the performance.

The experimental setup is the one presented in Chapter 2, Section 2.5.2 consisting of a CAKE platform with four TriMedia processor cores executing the applications $A_1, ..., A_6$. We investigate the compositionality and the performance of this system for L2 cache sizes varying from 256 KB to 2 MB, for different scenario switching rates ranging from from 100Hz (one switch every 0.01 second) to 1Hz (one switch every second). In the remainder of this section we first present the compositionality evaluation and then the performance figures.

6.4.1 Compositionality

To evaluate compositionality, we look at the critical task execution time variations in particular and at the number of inter-task conflicts in general.

To check the critical task execution time (et^{cr}) variation we simulate the

same application with random scenarios order, and different scenario switching rates, (1Hz to 100Hz). The critical task execution time variation, is defined, for a given execution, as the relative difference between the current execution time and the minimum execution time the critical task experienced, over all simulations:

$$\Delta(et^{cr}) = \frac{et^{cr} - \min\limits_{all\ execs.\ i}(et_i^{cr})}{\min\limits_{all\ execs.\ i}(et_i^{cr})} \tag{6.13}$$

		100Hz	50Hz	20Hz	10Hz	5Hz	1Hz	max. variation
A_1	Critical tasks prio	0.4%	1.3%	1.6%	1.7%	1.9%	0.0%	1.9%
	No critical tasks prio	1.5%	2.8%	1.8%	1.0%	0.2%	0.0%	2.8%
	Shared	1.7%	1.4%	1.7%	0.6%	0.0%	0.6%	1.7%
A_2	Critical tasks prio	1.1%	1.3%	2.2%	1.4%	1.4%	0.0%	2.2%
	No critical tasks prio	4.6%	4.6%	5.0%	5.1%	4.1%	0.0%	5.1%
	Shared	11.8%	11.1%	15.6%	7.4%	18.6%	0.0%	18.6%
A_3	Critical tasks prio	2.0%	1.5%	1.2%	0.8%	0.3%	0.0%	2.0%
	No critical tasks prio	3.8%	3.8%	0.2%	0.0%	0.2%	0.1%	3.8%
	Shared	5.2%	4.0%	3.4%	3.3%	0.0%	7.5%	7.5%
A_4	Critical tasks prio	0.7%	0.7%	1.0%	2.3%	0.4%	0.0%	2.3%
	No critical tasks prio	7.7%	16.2%	0.4%	0.8%	0.4%	0.0%	16.2%
	Shared	1.8%	1.3%	0.0%	0.9%	4.7%	3.5%	4.7%
A_5	Critical tasks prio	1.4%	2.1%	0.0%	1.8%	0.4%	1.9%	2.1%
	No critical tasks prio	3.7%	3.8%	3.3%	7.4%	4.0%	0.0%	7.4%
	Shared	4.2%	2.7%	3.5%	1.7%	0.0%	7.3%	7.3%
A_6	Critical tasks prio	1.4%	1.8%	0.0%	0.3%	2.0%	1.3%	2.0%
	No critical tasks prio	0.4%	13.3%	0.7%	0.0%	0.5%	2.8%	4.6%
	Shared	3.8%	1.0%	0.0%	1.1%	4.6%	2.4%	13.3%
avg	Critical tasks prio	1.1%	1.2%	0.7%	1.1%	0.8%	0.5%	1.2%
	No critical tasks prio	3.4%	6.9%	1.6%	2.2%	1.5%	0.5%	6.9%
	Shared	4.5%	3.4%	3.7%	2.4%	4.7%	3.5%	4.7%

Table 6.1: Critical tasks execution time and their variations (L2 size 256 KB).

In the Tables 6.1, 6.2, 6.3, and 6.4 we present the variation in critical tasks execution time corresponding to the 4 investigated L2 sizes (256 KB, 512 KB, 1MB and 2MB, respectively). We investigate three cases: (1) the cache footprints determined with the method in Section 6.2 (*Critical task prio*), (2) the cache footprints determined with the method in Section 6.2, but giving no priority to critical task (*No critical task prio*), and (3) the conventional shared cache (*Shared*). The graph in Figure 6.5 presents the maximum et^{cr} variations for all the investigated cache sizes, over all the experimented scenario switching rates. Concretely, from the Tables 6.1, 6.2, 6.3, and 6.4, we observed the following:

- for a cache size of 256 KB, in the case of *Critical task prio*, the maximum et^{cr} variation is 2.3%, for a switching frequency of 10 Hz, for the application

		100Hz	50Hz	20Hz	10Hz	5Hz	1Hz	max. variation
A_1	Critical tasks prio	1.8%	0.5%	0.4%	0.0%	0.1%	1.4%	1.8%
	No critical tasks prio	4.4%	2.3%	1.9%	1.1%	0.0%	0.3%	4.4%
	Shared	1.9%	3.0%	0.1%	0.0%	8.9%	13.8%	8.9%
A_2	Critical tasks prio	1.0%	0.0%	2.0%	0.4%	1.2%	1.6%	2.0%
	No critical tasks prio	0.9%	0.0%	1.1%	0.0%	0.2%	6.8%	6.8%
	Shared	4.2%	1.9%	2.5%	0.0%	4.1%	3.5%	4.2%
A_3	Critical tasks prio	0.7%	0.7%	0.1%	0.0%	0.2%	0.4%	0.7%
	No critical tasks prio	0.4%	0.4%	0.4%	0.3%	0.2%	0.0%	0.4%
	Shared	1.3%	1.9%	0.9%	0.5%	0.0%	0.0%	1.9%
A_4	Critical tasks prio	0.2%	0.0%	0.1%	0.0%	0.1%	0.0%	0.2%
	No critical tasks prio	0.3%	0.0%	0.0%	6.6%	0.1%	0.2%	6.6%
	Shared	1.4%	0.9%	1.2%	1.7%	0.0%	1.5%	1.7%
A_5	Critical tasks prio	0.7%	0.0%	1.2%	0.5%	1.7%	0.4%	1.7%
	No critical tasks prio	0.6%	0.6%	0.1%	8.2%	0.0%	0.1%	8.2%
	Shared	4.4%	4.5%	2.6%	3.2%	0.0%	10.3%	10.3%
A_6	Critical tasks prio	0.0%	0.2%	0.3%	0.2%	0.2%	0.1%	0.3%
	No critical tasks prio	0.3%	0.0%	0.4%	2.0%	0.2%	0.2%	2.0%
	Shared	1.4%	1.3%	1.8%	0.6%	0.7%	0.0%	1.8%
avg	Critical tasks prio	0.7%	0.2%	0.7%	0.2%	0.6%	0.4%	0.7%
	No critical tasks prio	1.1%	0.5%	0.6%	3.0%	0.1%	1.2%	3.0%
	Shared	2.4%	2.3%	1.5%	1.0%	2.3%	4.9%	4.9%

Table 6.2: Critical tasks execution time and their variations (L2 size 512 KB).

A_4, (with an average over all the applications, of maximum 1.2% at 5 Hz). In the case of *No critical task prio* the maximum variation reaches a value of 16.2%, for a switching frequency of 50 Hz, for the same A_4 (that, averaged over all the applications, has a maximum of 4.7% at 20 Hz). In the *Shared* cache case the maximum variation reaches a value of 18.6%, for a switching frequency of 5 Hz, for the application A_2 (with an average over all the applications, of maximum 6.9% at 5 Hz switching frequency).

- for a cache size of 512 KB, in the case of *Critical task prio*, the maximum et^{cr} variation is 2%, for a switching frequency of 20 Hz, for the application A_2, with an average over all the applications, of maximum 0.7% at two scenario switching frequencies of 1 and 10 Hz. In the case of *No critical task prio* the maximum variation reaches a value of 8.2%, for a switching frequency of 10 Hz, for the application A_5, with an average over all the applications, of maximum 3.0% at 10 Hz switching frequency. In the *Shared* cache case the maximum variation reaches a value of 10.3%, for a switching frequency of 1 Hz, for the application A_5, that, averaged over all the applications, has a maximum of 4.9% at 100 Hz.

- for a cache size of 1 MB, in the case of *Critical task prio*, the maximum et^{cr} variation is 2.2%, for a switching frequency of 10 Hz, for the application A_2, with an average over all the applications, of maximum 1.2% at 1 Hz

		100Hz	50Hz	20Hz	10Hz	5Hz	1Hz	max. variation
A_1	Critical tasks prio	1.9%	1.9%	1.4%	0.4%	1.2%	0.0%	1.9%
	No critical tasks prio	0.0%	4.0%	3.9%	4.0%	3.9%	2.4%	4.0%
	Shared	0.4%	1.3%	0.0%	0.6%	0.3%	4.0%	4.0%
A_2	Critical tasks prio	1.7%	2.0%	2.1%	2.2%	1.4%	0.0%	2.2%
	No critical tasks prio	4.8%	4.8%	5.2%	5.4%	4.3%	0.0%	5.4%
	Shared	7.7%	7.6%	8.3%	6.9%	7.1%	0.0%	8.3%
A_3	Critical tasks prio	1.1%	0.6%	0.0%	0.0%	0.3%	0.2%	1.1%
	No critical tasks prio	1.1%	1.4%	0.8%	0.3%	0.0%	0.7%	1.4%
	Shared	2.6%	2.2%	0.0%	0.4%	0.0%	1.3%	2.6%
A_4	Critical tasks prio	1.1%	0.0%	0.0%	0.5%	0.3%	1.7%	1.7%
	No critical tasks prio	5.9%	6.9%	6.8%	0.0%	6.2%	7.3%	7.3%
	Shared	0.7%	0.4%	0.1%	0.2%	0.0%	4.2%	4.2%
A_5	Critical tasks prio	0.1%	0.7%	0.9%	1.0%	0.0%	1.9%	1.9%
	No critical tasks prio	3.0%	3.8%	4.0%	4.0%	4.2%	0.0%	4.2%
	Shared	2.3%	1.8%	2.2%	0.6%	0.0%	0.6%	2.3%
A_6	Critical tasks prio	1.2%	0.0%	0.7%	0.7%	0.2%	0.5%	1.2%
	No critical tasks prio	1.4%	0.7%	0.8%	1.1%	0.4%	0.0%	1.4%
	Shared	1.9%	1.1%	0.0%	0.6%	0.7%	0.4%	1.9%
avg	Critical tasks prio	1.2%	0.9%	0.9%	0.8%	0.6%	0.7%	1.2%
	No critical tasks prio	2.7%	3.6%	3.6%	2.5%	3.2%	1.7%	3.6%
	Shared	2.6%	2.4%	1.8%	1.6%	1.3%	1.7%	2.6%

Table 6.3: Critical tasks execution time and their variations (L2 size 1 MB).

switching frequency. In the case of *No critical task prio* the maximum variation reaches a value of 7.3%, for a switching frequency of 1HZ Hz, for the application A_4, and, averaged over all the applications, it has a maximum of 3.6% at two scenario switching frequencies of 5 and 10 Hz. In the *Shared* cache case the maximum variation reaches a value of 8.3%, for a switching frequency of 20 Hz, for the application A_2, and, averaged over all the applications, it has a maximum of 2.6% at 1 Hz.

- for a cache size of 2 MB, in the case of *Critical task prio*, the maximum et^{cr} variation is 2.1%, for a switching frequency of 50 Hz, for the application A_3. In the case of *No critical task prio* the maximum variation reaches a value of 4.3%, for a switching frequency of 100 Hz, for the application A_6. In the *Shared* cache case the maximum variation reaches a value of 4.8%, for a switching frequency of 5 Hz, for the application A_5.

When comparing the *Critical task prio* case and the *No critical task prio* one we can see that, if critical tasks cannot tolerate variations larger than 3%, priority must be given to them at cache mapping stage. In all the presented experiments the variations in et^{cr} are very small for the *Critical task prio* case (representing at maximum only 2.3% from the critical tasks execution time). This is definitely not the case for *No critical task prio* and *Shared*. If no priority is given to the mapping of the critical tasks the et^{cr} variations increase,

		100Hz	50Hz	20Hz	10Hz	5Hz	1Hz	max. variation
A_1	Critical tasks prio	0.4%	1.5%	1.7%	0.0%	1.1%	2.1%	2.1%
	No critical tasks prio	2.4%	2.2%	3.1%	2.8%	0.0%	1.6%	3.1%
	Shared	1.6%	3.5%	3.2%	0.0%	2.3%	2.9%	3.5%
A_2	Critical tasks prio	1.6%	0.8%	0.5%	0.0%	1.6%	1.3%	1.6%
	No critical tasks prio	1.9%	1.7%	1.3%	2.0%	0.5%	1.7%	2.0%
	Shared	3.0%	4.1%	1.2%	0.0%	2.8%	1.7%	4.1%
A_3	Critical tasks prio	2.0%	2.1%	0.9%	0.8%	1.7%	0.0%	2.1%
	No critical tasks prio	1.5%	1.9%	2.5%	3.3%	1.2%	0.0%	3.3%
	Shared	2.2%	1.0%	2.5%	2.4%	0.0%	3.0%	3.0%
A_4	Critical tasks prio	0.7%	0.3%	0.0%	0.6%	1.6%	1.7%	1.7%
	No critical tasks prio	2.2%	3.4%	0.0%	1.5%	1.8%	2.6%	3.4%
	Shared	3.0%	2.8%	4.0%	3.5%	3.3%	0.0%	3.5%
A_5	Critical tasks prio	0.0%	1.7%	0.6%	0.3%	1.5%	1.9%	1.9%
	No critical tasks prio	1.1%	2.9%	0.5%	0.3%	0.0%	1.6%	2.9%
	Shared	3.5%	2.0%	2.3%	1.9%	4.8%	2.7%	4.8%
A_6	Critical tasks prio	0.3%	1.4%	0.0%	1.8%	1.5%	0.6%	1.8%
	No critical tasks prio	4.3%	2.1%	3.2%	0.0%	2.9%	1.4%	4.3%
	Shared	2.7%	0.0%	3.6%	4.2%	3.1%	2.5%	4.2%
avg	Critical tasks prio	1.2%	0.9%	0.9%	0.8%	0.6%	0.7%	1.2%
	No critical tasks prio	2.7%	3.6%	3.6%	2.5%	3.2%	1.7%	3.6%
	Shared	2.6%	2.4%	1.8%	1.6%	1.3%	1.7%	2.6%

Table 6.4: Critical tasks execution time and their variations (L2 size 2 MB).

reaching a relative value of 16.2%, with an overall average of 3.5%. For the shared cache the relative et^{cr} variations represent 18.6% from the minimum *Shared* et^{cr}, with an overall average of 3.3%. When comparing the *No critical task prio* case and the *Shared* one can see that, in general, our method, even with no priority given to the critical tasks, induced with up to 5% less maximum et^{cr} variation than the shared L2.

When looking at the maximum critical tasks execution time variations depending on scenario switching frequency we notice no clear correlation, as visible in Figure 6.5. This is somehow counter-intuitive, as one would expect that the critical tasks have their minimum execution time for small switching frequencies (visible as a 0% variation in the Table 6.1, 6.2, 6.3, and 6.4), and that the execution time increases (thus the variations to grow) with the switching frequency. The reason why the system is not behaving as the intuition tells is that the execution time is not dependent only on the number of flushes, but also on the moment when those happen. If, for example, the tasks execution is on a point between the processing of two frames when the flush occurs, the flushing penalty is dependent on the amount of data reuse (the dependencies) among the two frames.

In general we can observe that the critical tasks execution time variations decrease with the increase of the cache size, as directly visible in Figure 6.6.

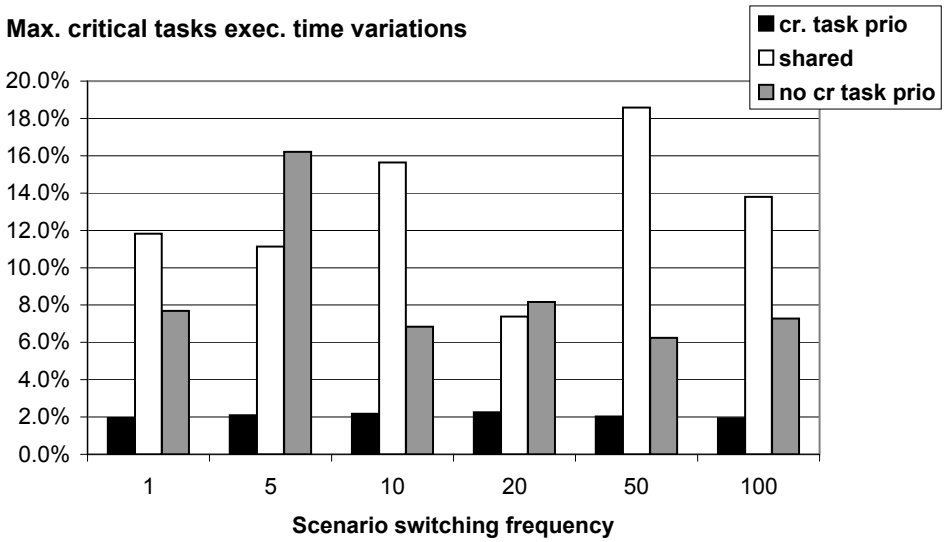

Figure 6.5: Maximum critical tasks execution time variations depending on the scenario switching frequency.

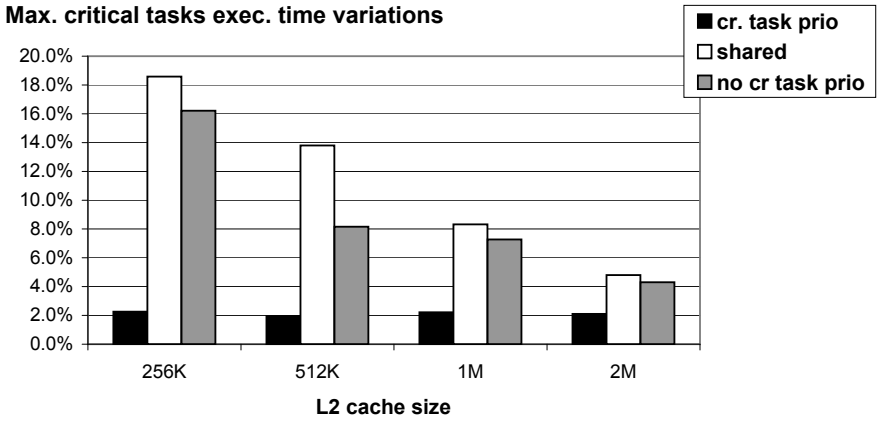

Figure 6.6: Maximum critical tasks execution time variations depending on the L2 size.

This is an expected effect because as the cache increases, more of the tasks' footprint fit in it (if the L2 is large enough the complete footprints will reside there). Therefore less data is swapped out at scenario switch, thus tasks' execution time variations decrease accordingly.

Apart from critical tasks execution time variations, the other metric used for evaluating the compositionality is the number of inter-task conflicts, as defined in Chapter 3, Section 3.5. These conflicts occur as a results of the late cache flush policy presented in the previous section. The Tables 6.5 and

		256 KB						512 KB					
		100Hz	50Hz	20Hz	10Hz	5Hz	1Hz	100Hz	50Hz	20Hz	10Hz	5Hz	1Hz
A_1	Shared	71%	73%	69%	66%	63%	66%	70%	62%	66%	53%	64%	56%
	St. part.	0%	0%	1%	0%	0%	0%	1%	0%	0%	0%	0%	0%
	Dyn. part.	8%	2%	1%	1%	1%	0%	3%	2%	0%	0%	0%	0%
A_2	Shared	81%	76%	73%	77%	75%	85%	75%	74%	73%	72%	76%	74%
	St. part.	0%	1%	0%	0%	0%	0%	0%	1%	0%	0%	1%	0%
	Dyn. part.	12%	7%	5%	0%	1%	0%	9%	8%	4%	4%	1%	0%
A_3	Shared	70%	71%	68%	68%	70%	73%	69%	69%	71%	63%	62%	59%
	St. part.	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
	Dyn. part.	4%	6%	3%	2%	1%	0%	4%	1%	2%	1%	0%	0%
A_4	Shared	76%	75%	73%	74%	72%	79%	77%	72%	73%	78%	77%	75%
	St. part.	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
	Dyn. part.	6%	1%	0%	0%	0%	0%	11%	3%	1%	0%	0%	0%
A_5	Shared	76%	77%	78%	78%	79%	78%	76%	78%	75%	74%	72%	70%
	St. part.	1%	0%	0%	1%	0%	0%	1%	0%	0%	0%	0%	0%
	Dyn. part.	5%	3%	2%	2%	1%	0%	6%	1%	0%	0%	0%	0%
A_6	Shared	79%	77%	69%	68%	73%	70%	77%	74%	72%	73%	72%	75%
	St. part.	0%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
	Dyn. part.	9%	7%	2%	0%	0%	0%	7%	5%	2%	1%	0%	0%
avg	Shared	76%	75%	72%	72%	72%	75%	74%	71%	71%	69%	71%	69%
	St. part.	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
	Dyn. part.	7%	4%	2%	1%	1%	0%	7%	3%	2%	1%	0%	0%

Table 6.5: Inter-task conflict misses per scenario change frequency (L2 sizes: 256 KB and 512 KB).

6.6 illustrate the relative number of conflicts for the 4 investigated L2 sizes, for each of the 6 applications, depending on the scenario switching frequency. These conflict misses are depicted for three cases: conventional shared L2 (*Shared*), static set based partitioned cache (*St. part*) and dynamic set based repartitioned L2 (*Dyn. part*). These values are relative to the corresponding application total number of misses. The last row represents an average value of all the applications inter-task conflicts. These experimental data lead to the following observations:

- For a cache size of 256 KB, in the case of *Shared* cache, the average number of conflicts reaches 74%, with a maximum of 81%, in the case of a switching frequency of 100 Hz, for A_2. In the *Dyn. part* cache case the maximum number of conflicts has a value of 12% (with an average of 3%), for a switching frequency of 100 Hz, for the application A_2.

- For a cache size of 512 KB, in the case of *Shared* cache, the average number of conflicts reaches 71%. In this case the maximum number of conflicts represents 77% from the total misses and it is observed for a switching frequencies of 100 Hz, for A_4, A_5 and A_6. In the *Dyn. part* cache case the number of conflicts peaks at a value of 11% (with an average of 2%), for a

		1 MB						2 MB					
		100Hz	50Hz	20Hz	10Hz	5Hz	1Hz	100Hz	50Hz	20Hz	10Hz	5Hz	1Hz
A_1	Shared	71%	69%	62%	65%	60%	61%	68%	65%	62%	62%	69%	62%
	St. part.	1%	1%	0%	0%	0%	0%	0%	0%	0%	1%	0%	0%
	Dyn. part.	9%	7%	2%	0%	0%	0%	2%	1%	0%	0%	0%	0%
A_2	Shared	67%	66%	66%	67%	66%	68%	63%	62%	61%	61%	59%	62%
	St. part.	1%	0%	1%	0%	0%	0%	1%	0%	0%	0%	0%	0%
	Dyn. part.	3%	2%	2%	0%	0%	0%	9%	2%	0%	0%	0%	0%
A_3	Shared	80%	80%	78%	74%	78%	78%	72%	68%	68%	75%	75%	69%
	St. part.	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
	Dyn. part.	8%	7%	1%	0%	1%	0%	3%	3%	1%	2%	1%	0%
A_4	Shared	76%	66%	65%	60%	62%	66%	58%	56%	55%	53%	52%	56%
	St. part.	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
	Dyn. part.	1%	1%	1%	0%	0%	0%	4%	1%	1%	0%	0%	0%
A_5	Shared	75%	73%	74%	72%	72%	74%	66%	65%	65%	63%	62%	64%
	St. part.	0%	0%	0%	0%	0%	0%	1%	0%	0%	0%	0%	0%
	Dyn. part.	1%	1%	1%	1%	0%	0%	8%	2%	0%	0%	0%	0%
A_6	Shared	71%	71%	72%	70%	71%	72%	66%	65%	63%	57%	61%	60%
	St. part.	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
	Dyn. part.	5%	1%	1%	1%	1%	0%	7%	6%	2%	0%	0%	0%
avg	Shared	73%	71%	70%	68%	68%	70%	66%	64%	62%	62%	63%	62%
	St. part.	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
	Dyn. part.	5%	3%	1%	0%	0%	0%	5%	2%	1%	0%	0%	0%

Table 6.6: Inter-task conflict misses per scenario change frequency (L2 sizes: 1MB and 2MB).

switching frequency of 100 Hz, for the application A_4.

- For a cache size of 1 MB, in the case of *Shared* cache, the average number of conflicts is 70%, with a maximum of 80%, in the case of the application A_3, for switching frequencies of 50 and 100 Hz. In the *Dyn. part* cache case the maximum number of conflicts has a value of 9% (with an average of 2%), for a switching frequency of 100 Hz, for the application A_1.

- for a cache size of 2 MB, in the case of *Shared* cache, the average number of conflicts reaches 63%, with a maximum of 72%, in the case of the application A_3, for a switching frequency of 100 Hz. In the *Dyn. part* cache case the maximum number of conflicts has a value of 9% (with an average of 1%), for a switching frequency of 100 Hz, for the application A_2.

Regardless of the cache size and the switching frequency, in the case of *St. part* the maximum number of conflicts has a value of 1%, as expected having in mind the results obtained in Chapter 3, Subsection 3.6.1 where we investigate the static cache partitioning compositionality.

As visible in Tables 6.5 and 6.6 and more concise in Figure 6.7, when the L2 is repartitioned and the scenarios are switched at a high frequency (20 Hz to 100 Hz), the average (over all applications and cache sizes) relative

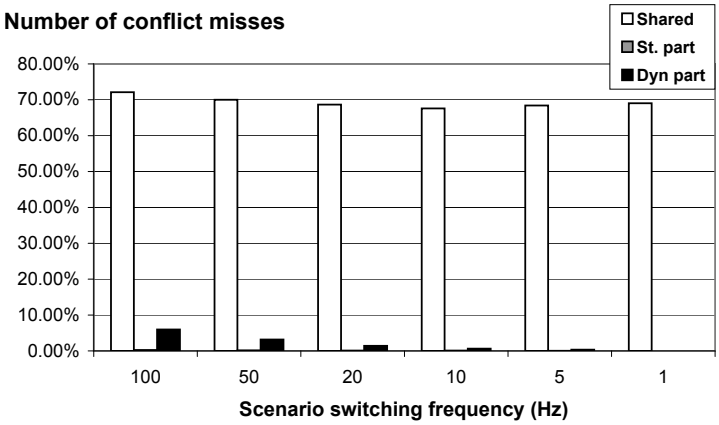

Figure 6.7: Number of conflict misses depending on the scenario switching frequency.

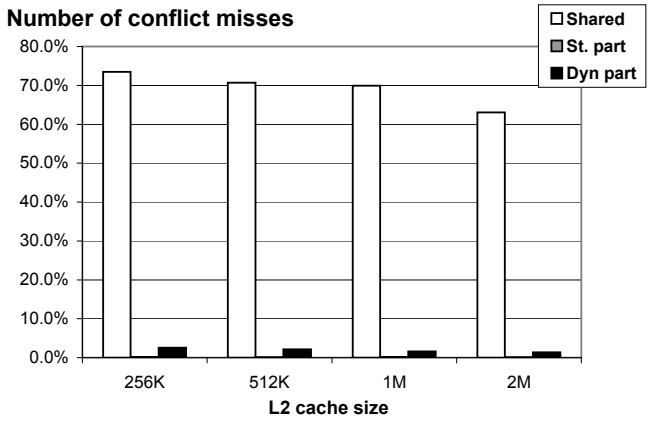

Figure 6.8: Number of conflict misses depending on the L2 size.

number of conflicts reaches a value of 3%. The maximum number of conflict misses encountered in these experiments is 12% for application A_2, for a 100 Hz scenario change frequency. For scenario switching rates under 10Hz the percentage of inter-task conflicts is at most 4% over all applications (with an average of 1%). As expected, the percentage of conflict misses decreases with the decrease of the scenario switch frequency, as the cache flushes occur less often.

When the cache size increases, for both the repartitioned and the shared cache, the number of conflicts tend to decrease, as visible in Figure 6.7. The

reason behind this phenomenon is that, in general, the number of cache misses is smaller for a larger cache, thus also the number of inter-task conflict misses decreases.

Unlike the partitioned cache, in a shared cache a large fraction of the misses represent actually inter-task conflict misses. The peak value for these misses is 81% and the average for all applications and all frequencies is 69%. Moreover, in this case there is no clear dependency among the scenarios switching frequency and the number of conflicts. The reason behind is that, unlike the repartitioned cache, the source of these conflict misses is not the cache flush required at scenario change (in the case of a shared cache, flushing is actually not needed). For a shared L2 the conflict misses occur not only at scenario switch, but during the entire execution and are caused by tasks freely swapping each other data (depending of input data, the moment of scenario switch, the execution history, etc.).

These results clearly suggest that the proposed dynamic repartitioning method results in a large improvement of the system compositionality, when compared with a conventional cache. When scenario switching happens less than 10 time per second the amount of inter-task conflicts is negligible ($< 4\%$), therefore we can consider that compositionality is achieved. Moreover, the critical tasks are practically undisturbed. Static cache partitioning scores close to ideal at compositionality, but, as one can see in the following subsection, this comes with a performance penalty.

6.4.2 Performance

We measure the performance using two metrics: (1) the number of misses per instruction (MPI), and (2) the processor's average cycles per instruction (CPI). We compare the performance in the following cases: (1) a set based repartitioned L2 with the cache footprints determined with Algorithm 2 presented in the previous section (*Alg2 footprints*), (2) a set based repartitioned L2 with optimally cache footprints, as determined in Section 6.2.2 (*Opt. footprints*), (3) a conventional shared L2 (*Shared*), (4) a statically set based partitioned L2 (*Static*), and (5) an infinite L2 cache (*Infinite*). The comparison with the performance of an infinite cache is interesting because it gives an idea about the maximum improvement that can be achieved by tuning the L2 cache. In our case it was enough to approximate an infinite L2 with a cache of 4 MBytes.

Tables 6.7, 6.8, 6.9, and 6.10 present the MPI and the CPI values for the 6 applications as functions of the scenario switching frequency, in the case of

an L2 of 256 KB, 512 KB, 1 MB, and 2 MB, respectively. The values in the Figures 6.9 and 6.10 represent the average over the 6 applications, for the 4 different L2 sizes. The MPI for the infinite cache is not presented as it is very close to 0. Figures 6.11 and 6.12 present the average MPI and CPI for the 6 applications as functions of the scenario switching frequency.

		MPI						CPI					
		1Hz	5Hz	10Hz	20Hz	50Hz	100Hz	1Hz	5Hz	10Hz	20Hz	50Hz	100Hz
A_1	dyn. part. Alg. 2	0.030	0.038	0.042	0.043	0.053	0.070	1.28	1.29	1.28	1.29	1.35	1.35
	dyn. part. Opt.	0.039	0.041	0.043	0.048	0.066	0.079	1.28	1.29	1.28	1.29	1.35	1.35
	st. part.	0.039	0.041	0.043	0.048	0.066	0.079	1.31	1.31	1.31	1.31	1.32	1.36
	shared	0.079	0.098	0.113	0.111	0.137	0.139	1.53	1.70	1.71	1.70	1.75	1.77
A_2	dyn. part. Alg. 2	0.072	0.079	0.092	0.092	0.102	0.147	1.32	1.37	1.42	1.42	1.45	1.55
	dyn. part. Opt.	0.077	0.079	0.092	0.092	0.122	0.166	1.32	1.37	1.42	1.42	1.45	1.55
	st. part.	0.095	0.097	0.099	0.102	0.108	0.128	1.41	1.48	1.47	1.50	1.50	1.53
	shared	0.109	0.128	0.134	0.139	0.146	0.160	1.61	1.81	1.82	1.82	1.84	1.86
A_3	dyn. part. Alg. 2	0.074	0.072	0.072	0.079	0.096	0.113	1.33	1.32	1.32	1.33	1.37	1.41
	dyn. part. Opt.	0.070	0.072	0.071	0.079	0.099	0.115	1.34	1.31	1.32	1.33	1.38	1.41
	st. part.	0.089	0.091	0.089	0.091	0.093	0.140	1.38	1.37	1.36	1.38	1.40	1.41
	shared	0.129	0.134	0.137	0.141	0.161	0.170	1.72	1.69	1.68	1.67	1.68	1.70
A_4	dyn. part. Alg. 2	0.019	0.020	0.021	0.023	0.021	0.024	1.19	1.19	1.19	1.20	1.25	1.24
	dyn. part. Opt.	0.019	0.019	0.021	0.023	0.021	0.022	1.19	1.19	1.19	1.21	1.25	1.23
	st. part.	0.036	0.041	0.039	0.038	0.038	0.038	1.28	1.34	1.31	1.28	1.28	1.30
	shared	0.047	0.046	0.045	0.050	0.049	0.052	1.52	1.50	1.54	1.57	1.54	1.54
A_5	dyn. part. Alg. 2	0.035	0.036	0.039	0.044	0.044	0.052	1.37	1.36	1.37	1.39	1.40	1.40
	dyn. part. Opt.	0.033	0.036	0.039	0.045	0.042	0.055	1.36	1.36	1.37	1.39	1.40	1.41
	st. part.	0.039	0.035	0.037	0.038	0.038	0.039	1.39	1.39	1.38	1.39	1.40	1.41
	shared	0.066	0.056	0.059	0.063	0.060	0.061	1.49	1.43	1.44	1.46	1.44	1.47
A_6	dyn. part. Alg. 2	0.094	0.103	0.107	0.110	0.111	0.129	1.34	1.37	1.40	1.39	1.41	1.45
	dyn. part. Opt.	0.092	0.101	0.108	0.109	0.107	0.127	1.34	1.36	1.40	1.39	1.40	1.45
	st. part.	0.131	0.142	0.143	0.151	0.151	0.166	1.47	1.53	1.53	1.56	1.58	1.61
	shared	0.158	0.175	0.162	0.209	0.227	0.257	1.58	1.59	1.67	1.69	1.76	1.78
avg	dyn. part. Alg. 2	0.054	0.058	0.062	0.065	0.071	0.089	1.31	1.32	1.33	1.34	1.37	1.40
	dyn. part. Opt.	0.055	0.058	0.062	0.066	0.076	0.094	1.31	1.31	1.33	1.34	1.37	1.40
	st. part.	0.072	0.075	0.075	0.078	0.082	0.098	1.37	1.40	1.39	1.40	1.41	1.44
	shared	0.098	0.106	0.108	0.119	0.130	0.140	1.58	1.62	1.64	1.65	1.67	1.69

Table 6.7: L2 size 256 KB: 100xMPI and CPI, per scenario change frequency.

Despite the flushing penalty, the MPI and CPI for the dynamic cache repartitioning using the proposed mapping solution is smaller than the case when the L2 is shared. On average over the 6 applications, and the scenario switching frequencies, these MPI and CPI reduction are as follows (per L2 size): (1) 47% and 18% for a 256 KB L2, (2) 37% and 12% for a 512 KB L2, (3) 26% and 6% for a 1 MB L2, and (4) 23% and 2% for a 2 MB L2. The above mentioned CPI improvements represent the following fractions from the possible improvements when having an ideal, infinite L2 (again per L2 size): (1) 35% for a 256 KB L2, (2) 60% for a 512 KB L2, (3) 51% for a 1 MB L2, and (4) 52% for a 2 MB L2. We would like to underline that these improvements occur

		MPI						CPI					
		1Hz	5Hz	10Hz	20Hz	50Hz	100Hz	1Hz	5Hz	10Hz	20Hz	50Hz	100Hz
A_1	dyn. part. Alg. 2	0.040	0.040	0.036	0.039	0.051	0.046	1.16	1.19	1.19	1.18	1.24	1.25
	dyn. part. Opt.	0.036	0.040	0.032	0.047	0.056	0.043	1.13	1.18	1.19	1.17	1.25	1.25
	st. part.	0.071	0.082	0.085	0.097	0.129	0.103	1.19	1.21	1.23	1.25	1.26	1.29
	shared	0.050	0.049	0.053	0.053	0.062	0.066	1.30	1.30	1.30	1.33	1.33	1.35
A_2	dyn. part. Alg. 2	0.015	0.029	0.028	0.037	0.047	0.064	1.24	1.18	1.25	1.24	1.28	1.25
	dyn. part. Opt.	0.013	0.026	0.031	0.037	0.047	0.061	1.24	1.18	1.25	1.23	1.27	1.24
	st. part.	0.017	0.032	0.038	0.040	0.038	0.046	1.25	1.18	1.25	1.25	1.26	1.24
	shared	0.065	0.063	0.058	0.063	0.062	0.064	1.38	1.38	1.39	1.40	1.41	1.41
A_3	dyn. part. Alg. 2	0.007	0.010	0.010	0.008	0.009	0.012	1.18	1.17	1.20	1.18	1.23	1.22
	dyn. part. Opt.	0.006	0.009	0.009	0.008	0.008	0.011	1.17	1.17	1.20	1.18	1.23	1.22
	st. part.	0.010	0.015	0.014	0.012	0.011	0.011	1.24	1.25	1.25	1.27	1.27	1.26
	shared	0.015	0.021	0.024	0.024	0.024	0.026	1.24	1.27	1.28	1.29	1.28	1.28
A_4	dyn. part. Alg. 2	0.008	0.008	0.007	0.009	0.008	0.008	1.26	1.26	1.32	1.31	1.30	1.29
	dyn. part. Opt.	0.007	0.007	0.007	0.009	0.008	0.008	1.27	1.26	1.32	1.31	1.30	1.30
	st. part.	0.008	0.008	0.007	0.010	0.009	0.009	1.32	1.32	1.36	1.32	1.29	1.29
	shared	0.009	0.010	0.011	0.011	0.010	0.011	1.49	1.41	1.40	1.43	1.41	1.41
A_5	dyn. part. Alg. 2	0.014	0.018	0.020	0.022	0.025	0.031	1.35	1.28	1.29	1.30	1.35	1.34
	dyn. part. Opt.	0.014	0.018	0.019	0.021	0.025	0.031	1.34	1.29	1.28	1.30	1.35	1.34
	st. part.	0.029	0.026	0.032	0.032	0.037	0.050	1.41	1.33	1.36	1.35	1.38	1.37
	shared	0.067	0.057	0.058	0.060	0.060	0.062	1.49	1.43	1.44	1.44	1.46	1.45
A_6	dyn. part. Alg. 2	0.042	0.043	0.035	0.039	0.039	0.040	1.10	1.12	1.16	1.18	1.28	1.31
	dyn. part. Opt.	0.042	0.041	0.035	0.040	0.043	0.050	1.10	1.12	1.16	1.19	1.28	1.32
	st. part.	0.078	0.058	0.046	0.047	0.045	0.052	1.17	1.18	1.23	1.22	1.30	1.31
	shared	0.101	0.097	0.097	0.105	0.101	0.101	1.36	1.39	1.37	1.37	1.38	1.39
avg	dyn. part. Alg. 2	0.021	0.019	0.023	0.026	0.030	0.033	1.22	1.19	1.23	1.23	1.27	1.27
	dyn. part. Opt.	0.020	0.018	0.022	0.027	0.031	0.034	1.21	1.20	1.23	1.23	1.28	1.28
	st. part.	0.036	0.025	0.037	0.040	0.045	0.045	1.06	1.24	1.28	1.27	1.29	1.29
	shared	0.051	0.050	0.050	0.052	0.053	0.055	1.38	1.36	1.36	1.37	1.38	1.38

Table 6.8: L2 size 512 KB: 100xMPI and CPI, per scenario change frequency.

when partitioning the L2 cache, while preserving its size. On average over all cache sizes, the improvement brought by the dynamic cache repartitioning is 33% in MPI and 10% in CPI, when compared to a shared L2.

In general the advantage of dynamic cache partitioning vs. the static one is that the dynamic partitioning scheme leads to a better utilization of the cache. The static scheme leaves a cache fraction not utilized, as not all task execute all the time, but they have a cache partition always allocated. However, the pitfall of dynamic set based repartitioning is that it requires cache flushing, which often comes with an overhead. The performance difference among the two partitioning schemes is given by the combination of these two factors, and is dependent of the application's cache access pattern and the scenario switching frequency. We experimentally observed that when comparing with a statically partitioned cache the dynamically partitioned one exhibits, on average over the 6 applications and the scenario switching frequencies, the following MPI and CPI reductions (per L2 size): (1) 27% and 4% for a 256 KB L2, (2) 26% and

4% for a 512 KB L2, (3) 17% and 2% for a 1 MB L2, and (4) 9% and 1% for a 2 MB L2. On average, the superiority of the dynamic over the static cache partitioning is represented by a 19% and 3% reduction in MPI and CPI, respectively.

		MPI						CPI					
		1Hz	5Hz	10Hz	20Hz	50Hz	100Hz	1Hz	5Hz	10Hz	20Hz	50Hz	100Hz
	dyn. part. Alg. 2	0.008	0.008	0.009	0.010	0.010	0.011	1.25	1.27	1.28	1.27	1.31	1.32
	dyn. part. Opt.	0.008	0.007	0.008	0.010	0.009	0.010	1.25	1.27	1.28	1.27	1.31	1.32
A_1	st. part.	0.014	0.016	0.021	0.024	0.025	0.025	1.28	1.29	1.32	1.32	1.33	1.36
	shared	0.010	0.009	0.013	0.013	0.012	0.016	1.26	1.26	1.27	1.30	1.30	1.33
	dyn. part. Alg. 2	0.035	0.038	0.041	0.039	0.038	0.038	1.16	1.22	1.23	1.23	1.23	1.18
	dyn. part. Opt.	0.035	0.053	0.041	0.035	0.052	0.033	1.16	1.22	1.23	1.23	1.23	1.18
A_2	st. part.	0.050	0.050	0.045	0.043	0.040	0.040	1.23	1.23	1.24	1.23	1.24	1.24
	shared	0.051	0.053	0.051	0.050	0.044	0.045	1.28	1.34	1.33	1.33	1.34	1.38
	dyn. part. Alg. 2	0.007	0.004	0.004	0.004	0.004	0.004	1.18	1.17	1.17	1.18	1.19	1.18
	dyn. part. Opt.	0.007	0.004	0.004	0.004	0.004	0.004	1.18	1.17	1.17	1.18	1.19	1.18
A_3	st. part.	0.006	0.004	0.005	0.005	0.005	0.005	1.18	1.17	1.18	1.18	1.19	1.19
	shared	0.007	0.006	0.005	0.005	0.007	0.006	1.20	1.19	1.18	1.19	1.20	1.19
	dyn. part. Alg. 2	0.001	0.001	0.001	0.001	0.001	0.001	1.19	1.16	1.16	1.16	1.17	1.18
	dyn. part. Opt.	0.001	0.001	0.001	0.001	0.001	0.001	1.19	1.16	1.16	1.16	1.16	1.16
A_4	shared	0.001	0.001	0.001	0.001	0.001	0.001	1.21	1.16	1.16	1.16	1.19	1.19
	shared	0.003	0.003	0.003	0.003	0.002	0.002	1.25	1.17	1.18	1.17	1.20	1.21
	dyn. part. Alg. 2	0.023	0.022	0.022	0.023	0.027	0.023	1.16	1.16	1.16	1.16	1.16	1.16
	dyn. part. Opt.	0.022	0.020	0.021	0.020	0.024	0.021	1.16	1.15	1.15	1.16	1.16	1.15
A_5	st. part.	0.024	0.021	0.023	0.023	0.027	0.025	1.16	1.17	1.16	1.17	1.17	1.17
	shared	0.034	0.031	0.032	0.034	0.032	0.034	1.31	1.28	1.28	1.29	1.29	1.29
	dyn. part. Alg. 2	0.014	0.014	0.014	0.014	0.014	0.016	1.10	1.11	1.11	1.10	1.12	1.10
	dyn. part. Opt.	0.013	0.015	0.013	0.014	0.014	0.015	1.11	1.10	1.10	1.10	1.11	1.10
A_6	st. part.	0.014	0.012	0.014	0.013	0.014	0.014	1.10	1.11	1.11	1.10	1.12	1.11
	shared	0.011	0.010	0.010	0.010	0.012	0.011	1.22	1.21	1.21	1.23	1.23	1.24
	dyn. part. Alg. 2	0.015	0.015	0.015	0.015	0.016	0.016	1.17	1.18	1.19	1.18	1.20	1.19
	dyn. part. Opt.	0.014	0.017	0.015	0.014	0.018	0.014	1.18	1.18	1.18	1.18	1.19	1.18
avg	st. part.	00.018	0.017	0.018	0.018	0.019	0.019	1.19	1.19	1.20	1.19	1.21	1.21
	shared	0.020	0.019	0.019	0.019	0.018	0.019	1.25	1.24	1.24	1.25	1.26	1.28

Table 6.9: L2 size 1 MB: 100xMPI and CPI, per scenario change frequency.

The performance differences in MPI and CPI among the static and dynamic partitioned cache decrease with the increase of scenario switching frequency. Let us look at the smallest L2 size used for the experiments (i.e.256 and 512 KB). For a scenario switching rate of 100Hz the dynamically partitioned L2 outperforms the statically partitioned one on average with 2% (with a maximum of 5%) for the CPI metric and with an average of 17% (with a maximum of 36%) for the MPI metric, whereas for a scenario switching rate of 1Hz the improvement reaches on average 5% (with a maximum of 7%) and 35% (with a maximum of 44%), respectively. For a cache size of 512 KB, for a scenario switching rate of 100Hz the dynamically partitioned L2 outperforms the statically partitioned one on average with 1% (with a maximum of 5%) for the CPI

		MPI						CPI					
		1Hz	5Hz	10Hz	20Hz	50Hz	100Hz	1Hz	5Hz	10Hz	20Hz	50Hz	100Hz
A_1	dyn. part. Alg. 2	0.005	0.005	0.006	0.007	0.007	0.007	1.24	1.26	1.26	1.27	1.27	1.27
	dyn. part. Opt.	0.005	0.005	0.005	0.007	0.006	0.007	1.25	1.25	1.26	1.26	1.26	1.27
	st. part.	0.009	0.011	0.014	0.016	0.017	0.017	1.25	1.26	1.26	1.25	1.26	1.27
	shared	0.007	0.006	0.009	0.009	0.008	0.011	1.26	1.26	1.27	1.26	1.26	1.26
A_2	dyn. part. Alg. 2	0.024	0.026	0.028	0.027	0.027	0.026	1.14	1.15	1.15	1.16	1.16	1.15
	dyn. part. Opt.	0.024	0.038	0.030	0.024	0.036	0.024	1.16	1.16	1.15	1.16	1.16	1.16
	st. part.	0.036	0.034	0.032	0.031	0.028	0.028	1.15	1.15	1.16	1.16	1.16	1.16
	shared	0.035	0.036	0.035	0.035	0.030	0.031	1.16	1.15	1.15	1.16	1.16	1.16
A_3	dyn. part. Alg. 2	0.003	0.002	0.002	0.002	0.002	0.002	1.18	1.17	1.17	1.17	1.19	1.18
	dyn. part. Opt.	0.003	0.002	0.002	0.002	0.002	0.002	1.17	1.17	1.17	1.18	1.19	1.18
	st. part.	0.003	0.002	0.002	0.002	0.002	0.002	1.19	1.18	1.19	1.19	1.19	1.19
	shared	0.017	0.014	0.009	0.009	0.011	0.011	1.19	1.18	1.19	1.19	1.19	1.19
A_4	dyn. part. Alg. 2	0.001	0.001	0.001	0.001	0.001	0.001	1.16	1.16	1.16	1.16	1.17	1.18
	dyn. part. Opt.	0.000	0.001	0.001	0.000	0.001	0.001	1.17	1.16	1.16	1.16	1.16	1.16
	st. part.	0.001	0.001	0.001	0.001	0.001	0.001	1.16	1.17	1.17	1.17	1.19	1.19
	shared	0.002	0.002	0.002	0.002	0.001	0.001	1.16	1.16	1.16	1.17	1.18	1.16
A_5	dyn. part. Alg. 2	0.015	0.014	0.014	0.015	0.017	0.015	1.15	1.15	1.15	1.16	1.15	1.16
	dyn. part. Opt.	0.015	0.014	0.014	0.013	0.014	0.013	1.16	1.16	1.16	1.15	1.16	1.16
	st. part.	0.016	0.015	0.015	0.014	0.019	0.017	1.16	1.16	1.16	1.16	1.16	1.16
	shared	0.022	0.030	0.021	0.020	0.021	0.022	1.16	1.14	1.15	1.15	1.15	1.15
A_6	dyn. part. Alg. 2	0.004	0.004	0.004	0.004	0.005	0.005	1.10	1.10	1.10	1.10	1.10	1.11
	dyn. part. Opt.	0.004	0.004	0.004	0.004	0.005	0.004	1.10	1.10	1.10	1.10	1.10	1.11
	st. part.	0.004	0.003	0.004	0.004	0.004	0.004	1.10	1.10	1.11	1.10	1.10	1.10
	shared	0.003	0.003	0.003	0.003	0.004	0.003	1.10	1.10	1.10	1.10	1.10	1.10
avg	dyn. part. Alg. 2	0.008	0.008	0.009	0.009	0.009	0.009	1.16	1.17	1.17	1.17	1.17	1.18
	dyn. part. Opt.	0.008	0.010	0.009	0.008	0.010	0.008	1.17	1.17	1.17	1.17	1.17	1.17
	st. part.	0.011	0.011	0.011	0.011	0.012	0.022	1.17	1.17	1.18	1.17	1.18	1.18
	shared	0.014	0.015	0.013	0.013	0.012	0.013	1.17	1.17	1.17	1.17	1.17	1.17

Table 6.10: L2 size 2 MB: 100xMPI and CPI, per scenario change frequency.

metric and with an average of 23% (with a maximum of 56%) for the metric MPI, whereas for a a scenario switching rate of 1Hz the improvement reaches on average 7% (with a maximum of 10%) and 47% (with a maximum of 56%), respectively. The performance differences among the static and dynamic partitioning scheme become smaller for larger cache sizes. This is an expected effect, as a large cache can contain most of the application's footprint, thus the L2 behaves close to ideal. As a result, in whatever manner one would tune the L2, it can have a little influence on the application's performance.

The only four cases when the dynamic partitioning performs worse than the static one is when the scenarios switch with a 100Hz rate for A_2 and A_3 with a cache of 512KB and for A_2 and A_5 with a cache of 256KB. Concretely, when compared to an dynamically partitioned cache, for an L2 of 256KB the statically partitioned cache causes a 14% better MPI, and 1% CPI for A_2, and 30% better MPI, and 1% CPI for A_5, respectively. For an L2 of 512KB the performance difference are 39% in MPI, and 1% CPI in for A_2, and 3% in MPI,

and no CPI difference for A_3, respectively. The reason behind this performance loss is that for these applications when scenarios are switched very fast the cache flushing penalty exceeds the benefits of using a larger part of the cache.

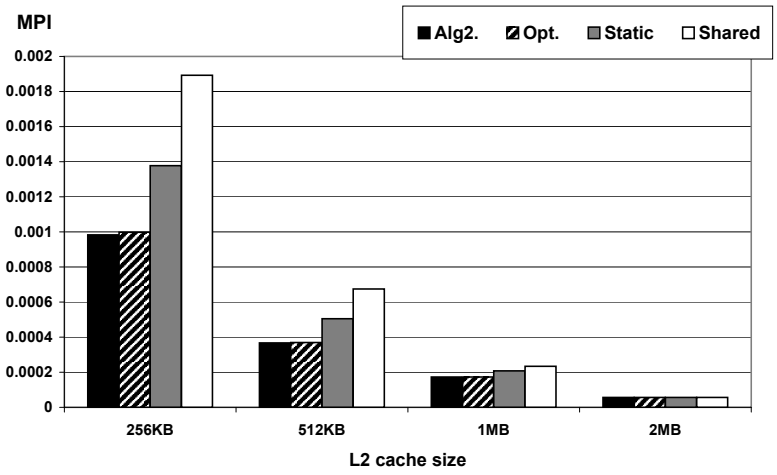

Figure 6.9: Performance: MPI depending on the L2 size.

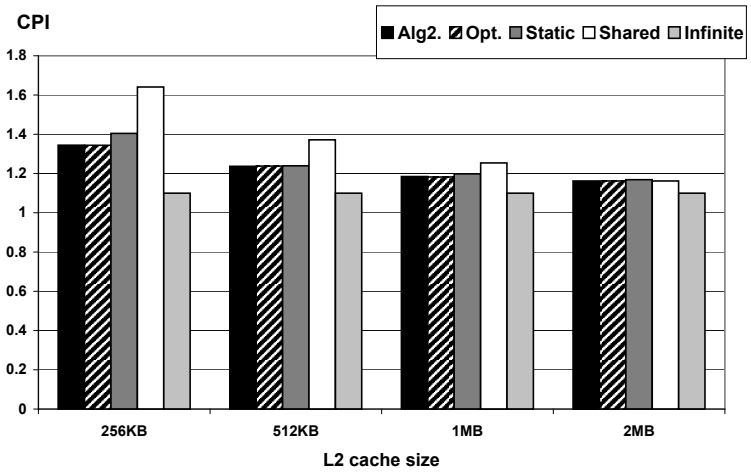

Figure 6.10: Performance: CPI depending on the L2 size.

When comparing the *Alg2 footprints* case with the optimally cache footprints, we can notice that the performance difference among the two is not large. At maximum, for a cache size of 256KB, *Opt. footprints* has 8% better MPI, resulting in 1% CPI improvement. Overall, on average the *Opt. foot-*

prints delivers 4% MPI improvement and very small CPI reduction, while, due to the large number of ILP variables required, its optimization time can be up to few days. Contrarily, the *Alg2 footprints* optimization time is in the order of magnitude of minutes. Therefore we believe that the small extra performance that *Opt. footprints* brings does not justify the large amount of time spend in optimization, and we conclude that the method in *Alg2* is more suitable for our case.

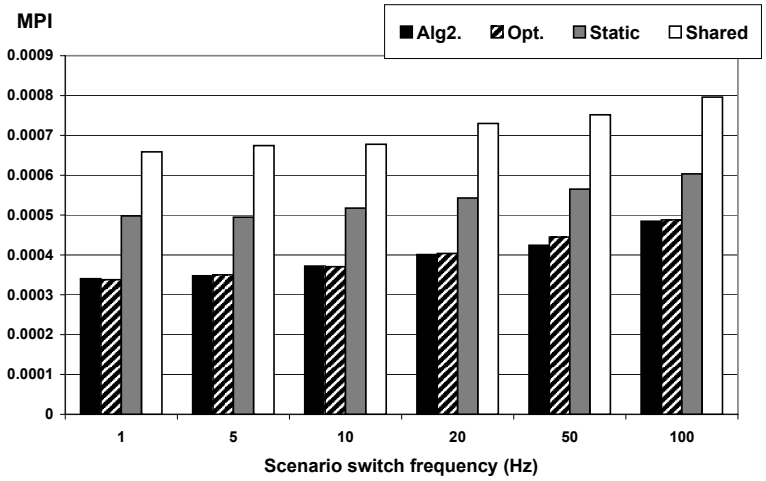

Figure 6.11: Performance: MPI depending on the scenario switching frequency.

The above mentioned figures clearly indicate that the use of the dynamic partitioning method in applications with multiple utilization scenarios, especially for low scenario switching rates (this rate is likely to be even lower than one switch every second), can be beneficial.

When looking solely at the dynamic partitioning involving Algorithm 2, we can notice that the average MPI and CPI increase when the scenario switching frequency is varied from 1Hz to 100Hz. For a cache of 256KB, the MPI increase when the scenario switching frequency is 1Hz is 34% larger than the one when switching the scenarios at 100Hz, resulting in a 4% CPI variation. The corresponding values for the other investigated cache sizes: 27% and 19% for 512KB, 5% and 1% for 1MB and for 2MB 3% and 0%. However, for realistic scenario switching ranges (over 10Hz) the difference is at maximum 20% in MPI and 2% in CPI. This suggests that in such a case the cache flushing penalty is not significant.

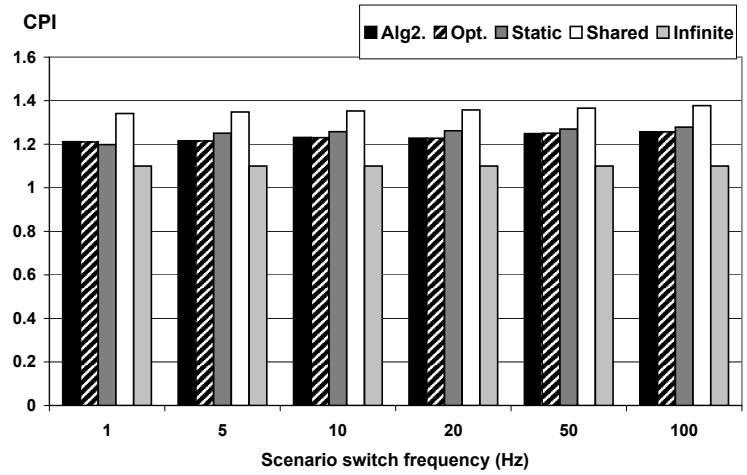

Figure 6.12: Performance: CPI depending on the scenario switching frequency.

6.5 Conclusion

In this chapter we proposed a dynamic cache management method that enhances compositionality for multimedia applications with multiple utilization scenarios. At run-time, when the scenario changes, tasks might start or stop, thus the cache is repartitioned. As the partitioning is set-based, the dynamic repartitioning requires L2 cache flushing in order to keep the data consistency. This involves an overhead that, in principle, negatively affects the system performance in general and the critical tasks behavior in particular. In this context we propose a method which: (1) at design-time determines the cache footprint of each tasks, such that the critical tasks are guaranteed to be undisturbed, and the flushing overhead is minimized in general, and (2) at run-time ensures that the cache footprints are enforced and further decreases the flush penalty. On a CAKE multiprocessor with 4 cores we investigated the compositionality and the performance induced by the proposed cache repartitioning over a wide range of scenario switching frequency (100Hz to 1Hz).

With respect to compositionality, for realistic scenario switching frequencies, we found that, relative to the application number of misses, the inter-task cache flushes are under 4% for the repartitioned cache, whereas for the shared cache they reach 81%. Moreover, the relative variations of critical tasks execution time are less than 0.1%, over the entire scenario switching frequency range studied, when the L2 is repartitioned according to our method. The

relative variations of critical tasks execution time reach a value of 18% for a shared cache, and 16% for a repartitioned cache that does not give priority to critical task when placing them in cache. Thus if critical tasks cannot tolerate variations larger than 3%, priority must be given to them at cache mapping stage.

Concerning the performance, the dynamic repartitioning reduces the number of L2 misses per instruction on average with 33% (and a peak of 47%), when compared with the shared cache. As a consequence, the average number of cycles needed to execute an instruction is decreased with as much as 18% (on average with 10%), when compared with the shared cache, under the circumstances that a maximum of 35% reduction is potentially achievable when using an infinite L2 cache. Thus 56% of the possible maximum improvement is achieved by repartitioning, while preserving the same L2 size. As expected, the performance of the shared and the repartitioned cache come closer and closer with the increase of the cache size (large caches can contain more of the application footprint, thus the total application performance is not that sensitive to cache optimizations of any kind, as it is for a small cache). Furthermore, when comparing with a statically partitioned cache, the dynamic repartitioning reduces the number of L2 misses per instruction on average with 19% (and a peak of 56%). The average number of cycles needed to execute an instruction is decreased with maximum 10% (on average with 3%), when compared with the same statically partitioned cache. Moreover, the misses per instruction difference among an optimal dynamic partition and the heuristic we propose is at maximum 4%. As the time to find the optimal partitioning may be up to several days, whereas the heuristics takes minutes, the small gained performance does not justify the large amount of time spend in optimization.

In conclusion, despite the involved cache flushing, the repartitioned L2 enables high compositionality and delivers better performance than the shared and the statically partitioned cache.

Chapter 7

Conclusions and future work

In this thesis we proposed a cache memory management method suited for embedded chip multiprocessors executing multimedia applications. Typical requirements in this domain are high performance, predictability, low resource usage, low cost, fast time to market, and dependability. In this context, flexibility is a key design requirement that facilitates the update to new standards and the addition of new features. Moreover, parts of the system can be easily reused, providing better premises for a fast time to market and a low production cost. On the other hand, accurate performance prediction for a flexible memory hierarchy requires a large analysis effort, which obstructs the time to market. The purpose of this dissertation was to obtain a suitable flexibility-predictability trade-off for media applications running on embedded multiprocessors. Predicting the performance of a parallel multimedia system involves predicting the performance of its sequential parts (called tasks), as well as their degree of interaction. As this interaction is typically unpredictable, especially in systems with caches, the performance of each sequential task should be (ideally) preserved if the tasks are executed concurrently in arbitrary combinations, or if additional tasks are added. A system satisfying this property is called compositional. In our scheme, we use hardware controlled memories (caches) for flexibility reasons and we proposed to use a task centric cache management scheme to ensure compositionality, thus preserving predictability. In this chapter, we first present the summary of this dissertation, then discuss the main contributions of this work, and finally suggest future promising research directions.

7.1 Summary

In Chapter 2, we first introduced the necessary terminology and definitions related to the desired properties of multimedia embedded systems, namely predictability, flexibility, and compositionality. We have also described the targeted platform (the CAKE multiprocessor) and the benchmark application suite (multimedia, multitasking) utilized in all the experiments presented in the rest of this thesis. The CAKE platform contains four media processor cores (TriMedia) and one control processor (MIPS), each one having its local first level of cache (L1 cache). Moreover, the platform contains a large, shared, second cache level (L2 cache). The applications executed on this platform consist of multiple tasks that may share instructions and/or data. More precisely, such an application may consist of independent tasks (from MediaBench augmented with an H.264 codec), or of communicating tasks (H.264 and PiPTV). These applications may be static, in the sense that tasks do not start and/or stop at run-time, or dynamic, if different execution scenarios (i.e., tasks do start and stop at run-time) may occur.

In Chapter 3 we introduced the task centric memory management concept. The main idea is to partition the shared L2 cache among the application tasks, such that these tasks do not conflict in the cache, and the system is compositional. Starting from a conventional cache organization, we identified two options for cache partitioning, namely set-based and associativity-based partitioning. Further, we proposed a technique to find the cache partitioning ratio that minimizes the overall number of misses (based on a Dynamic Programming formulation). We investigated the compositionality and the performance for these two options, as well as for the conventional shared cache, for various cache sizes of the CAKE platform. We proposed to use the number of inter-task conflicts misses as a metric for compositionality. The experiments indicated that both cache partitioning methods are potential candidates to keep the cache accesses of each task isolated and to induce compositionality, as the conflict misses for the partitioned cache represent less than 1% from the total application misses.

As expected, the conventional shared cache has poor compositionality: we found that, on average, inter-task conflict misses reach 66% of the application misses. Regarding the performance, we found that for 89% of the cache configurations (64 of the 72 size/associativity combinations we experimented with, the set-based cache management led to an improvement in cache performance (measured in misses per instruction, MPI) and to faster computation (measured in cycles per instruction, CPI), when compared with a shared cache.

More precisely, when compared to a shared cache, the set-based partitioned cache provides up to 62% less MPI and 31% less CPI (note that less MPI and CPI translate to better performance), with an average decrease of 29% and 8%, respectively. In only 8 cases out of the 72 investigated, we found that set-based partitioning degraded the cache performance. On average over these 8 cases, the performance degradation was 19% and 4% for MPI and CPI, respectively. We also found that most of the time (67 from the 72 cases), associativity-based partitioning degrades the memory hierarchy performance. When compared with the conventional cache organization, associativity-based partitioning increases the MPI with up to 187%, slowing down the computation with 229%, with an average penalty of 52% more MPI and 25% more CPI. Summarizing, the superiority of set-based cache partitioning over associativity-based partitioning is caused by two reasons: (1) the available number of allocable cache parts is larger, and (2) it does not reduce the associativity of a task's cache part. As set-based cache partitioning leads to far better performance, we have considered it the foundation for our task centric cache management.

In Chapter 4 we proposed a mixed, set and associativity-based cache partitioning method that ensures performance compositionality and also allows cache sharing for common tasks data and/or instructions (shortly named "common regions"). This method removes the inter-task cache interference by using two static cache partitioning types. First, each task and each inter-task common region gets allocated an exclusive part of the cache sets. Second, inside the cache sets of a common region, each task accessing it receives a number of ways. Furthermore, we add to the mixed partitioning method two techniques to determine the cache partitioning ratio. The first aproach is an extension of the method introduced in Chapter 3 (the method used to minimize the overall number of misses) that also supports sharing cache parts for data and instructions. The second method aims to optimize the application throughput and it is based on simulated annealing. In the case of throughput optimization, at every step of the annealing, the throughput of the system has to be estimated very fast. Therefore, we design and use a light simulation strategy. When compared with a cycle-true simulation, our novel, light simulation is 30 times faster and its accuracy is within 3%.

We have applied these techniques to two multi-tasking applications that share data and instructions (H.264 and PiPTV). On the CAKE platform, the experimental results indicated that the proposed cache partitioning ensures compositionality to a large extent, as for both applications the conflict misses represent less than 1% of the total misses. Moreover, we compared the performance of the following four cases: (1) the cache is fully shared, (2) the cache

is partitioned such that the number of misses is minimized (CPR_M), (3) the cache is partitioned such that the throughput is maximized (CPR_T), and (4) the cache is shared, but has a virtually infinite L2 (enough to contain the entire application's footprint - 8MB in the present case). We measured the L2 performance (in number of L2 misses per instruction - L2 MPI) and the application performance (in number of cycles per instruction - CPI, and throughput). As expected, the performance differences among the four cases are dependent on the L2 cache size. When the platform was equipped with a relatively small L2 cache size, our simulations indicate that the smallest number of misses achievable by a partitioned cache is larger than the one of a shared cache of the same dimensions. As both the PiPTV and H.264 applications have a large number of tasks and common regions, for small caches, the fragmentation of the cache entailed by partitioning caused a performance decrease of the CPR_M when compared with the shared cache. More precisely, on average over the two applications, the MPI increases with 17%, and the CPI with 6%. However, for the same sizes, the CPR_T case improves the throughput at the expense of extra misses. When compared with CPR_M, the CPR_T increases the throughput with 7% on average, under the circumstances that a maximum throughput increase of 25% is potentially achievable with an infinite L2 cache. Thus, with less than a quarter of the L2, our method achieves more than a third of the maximum throughput improvement reachable with an L2 cache of 8MB. This throughput increase corresponds to an L2 MPI increase with 14%. For medium cache sizes, the elimination of inter-task misses caused by partitioning supersedes the effect of cache fragmentation. The CPR_M partitioned L2 has 5% better CPI and 28% better MPI than the shared cache, on average. The CPR_T has more or less the same performance as the CPR_M. Finally, our experiments indicate that, for our applications, both CPR methods deliver compositionality, but speedup is achieved by the CPR_T optimization for small-size L2 caches, and by the CPR_M optimization for larger L2 caches.

In Chapter 5 we proposed a method to analyze the static cache partitioning robustness. Two types of robustness were discussed: internal (determined by inter-task interference in the L1 cache) and external (determined by the variations of the L2 behavior due to various input data sets). We have introduced quantification metrics for both robustness types. To assess internal robustness, we defined the sensitivity function, which measures the deviation of L2 misses caused by the L1 misses variations over a range of task switching rates. To assess the external robustness, we introduced the stability function, which measures the performance deviation for the case the application processes another input data set than the one utilized to determine the static L2

partitioning ratio. To demonstrate our approach, we analyzed both types of parallel applications introduced in Chapter 2 (i.e., consisting of independent tasks or of communicating tasks). Concerning the internal robustness, if the cache is partitioned, the application sensitivity is at most 8%, with an average of 4%. This small sensitivity reinforces the conclusion that partitioning the L2 is enough to achieve compositionality in a large degree, for these applications. Comparing the internal robustness of the shared and partitioned cache cases, we found that the shared cache is on average 6 times more sensitive than the partitioned one. Moreover, the large difference between the sensitivity of the shared and the partitioned caches is an interesting fact in itself, as it suggests that the optimization processes for L1 and L2 caches can be decoupled if the L2 is managed in a task centric manner. For the external robustness, the variations induced in the L2 behavior by various input data sets are at most 10% over the entire range of applications we have tried. This accounts for an average stability of 92%, therefore, for the investigated applications, we can conclude that the static cache partitioning is quite robust with respect to input stimuli variations.

In Chapter 6 we proposed a dynamic task centric cache management strategy. This strategy is tailored to multimedia application with multiple execution scenarios. In this case, during the transition between consecutive execution scenarios, some tasks may start and/or stop. If the cache is statically allocated, the part corresponding to a task would be reserved all the time for it, even when the task is not executing. In this manner, the cache is wasted and the system performance may be penalized. To avoid this waste, we propose a strategy to dynamically repartition the cache at a scenario change, such that we enhance compositionality and we efficiently utilize the entire cache. Due to compositionality, once the performance of each task is known, the cache partitioning ratio can be easily computed off-line for each scenario. At run-time, when the scenario changes, the cache is repartitioned to a new ratio. As the partitioning is set-based, preserving data correctness requires some cache flushing. This involves an overhead that, in principle, decreases the system performance in general and the critical tasks behavior in particular. In this context, we proposed a method which: (1) determines at design-time the cache footprint of each task, such that the critical tasks are guaranteed to be undisturbed and the flushing overhead is minimized, and (2) ensures at run-time that the cache footprints are enforced, while further decreasing the flush penalty. On the same CAKE multiprocessor, we investigated the compositionality and the performance induced by the proposed dynamic cache repartitioning over a wide range of scenario switching frequency (100Hz to 1Hz).

To assess compositionality, we have measured the number of inter-task cache misses relative to an application's total number of misses, for realistic scenario switching frequencies. We found that the inter-task cache misses are below 4% for the repartitioned cache, while they can reach 81% for the shared cache. Moreover, when the L2 is repartitioned according to our method, the relative variations of the critical tasks execution times are less than 0.1% for the entire scenario switching frequency range we have studied.

Regarding performance, our dynamic repartitioning reduces the L2 MPI with 33% on average (with a peak of 47%), when compared with the shared cache. As a consequence, for the dynamically repartitioned cache, the average number of cycles per instruction is 10% lower on average (with a peak of 18%) than for the shared cache; note that a maximum of 35% CPI reduction is potentially achievable when using an infinite L2 cache. Thus, using our dynamic repartitioning, we reach 56% of the maximum improvement, while preserving the L2 size. As expected, the performance difference between the shared and the dynamically repartitioned cache reduces with the increase of the cache size.

7.2 Main contributions

In the context of compositional memory hierarchies for embedded multiprocessors, the main contributions of this work can be listed as follows:

- We compared the two natural manners of partitioning a cache, namely set and associativity-based, and we found that both partitioning schemes can ensure compositionality within 1% bounds.
- We proposed a new set-based partitioning implementation method which does not require compiler modifications, and is not dependent of the memory addressing model.
- We proposed a technique to find the cache partitioning ratio that has the minimum number of application cache misses (based on a Dynamic Programming formulation).
- For media applications consisting of independent tasks from the MediaBench suite augmented with an industrially relevant H.264 codec, on a CAKE platform with 4 TriMedia processors, we found that the L2 misses per instruction resulting from set-based partitioning are on average 55% smaller than the ones corresponding to associativity-based

partitioning, and 29% smaller than the L2 misses per instruction of the conventional shared cache. This leads to an average application speedup of 27% and 8%, when compared to associativity-based partitioning and to a conventional shared cache, respectively. These facts recommend set-based partitioning as the best candidate for a task centric cache management scheme for multimedia applications on a multiprocessor.

- We introduced a new task centric cache management method tailored for media applications that share data and/or instructions. This method consists of a mixed (set- and associativity-based) cache partitioning scheme and two cache partitioning ratio optimization strategies: (1) minimize the number of misses and (2) maximize the throughput.

- By applying the our new method to the benchmark applications consisting of communicating tasks, we observed that the amount of inter-task interference is under 1%, indicating that the proposed cache partitioning ensures compositionality to a large extent.

- For applications consisting of communicating tasks, when the platform was equipped with a relatively small cache, our simulations indicated that the smallest number of misses achievable by a partitioned cache is larger than the one of a shared cache of the same dimensions. For the same (small) cache sizes, we observed that, when the throughput maximization method is applied, the throughput improves (with 7% on average) at the expense of misses growth (with 14% on average). Thus, for our applications, when using small caches, the price of compositionality is a degradation in performance. For average cache sizes, the partition that minimizes the number of misses also decreases both the CPI (with 5% on average) and MPI (with 28% on average) when compared with the shared cache. The partition that maximizes the throughput has more or less the same performance as the one minimizing the misses. For large cache sizes, the performance of the partitioned and shared cache are very close. Thus, for average size and large caches, mixed partitioning induces compositionality and always preserves or improves cache performance.

- To investigate the robustness of a system that is using our cache management we introduce two metrics to assess: (1) the performance deviations caused by the tasks comprising the application (internal robustness) and (2) performance variations caused by external stimuli (external robustness). According to our simulations, the system variations due to inter-

nal interferences are below 8% and the variations due to external factors are below 10%. Thus, the system is robust in the presence of cache partitioning.

- We extended the task centric cache management with a dynamic method for applications with multiple execution scenarios, and critical tasks. The dynamic partitioning ensures high cache utilization, exhibits high compositionality, and safeguards critical tasks against performance disruptions.

- We found that, for realistic scenario switching frequencies, relative to the application number of misses, the inter-task cache misses are below 4% for the repartitioned cache, while reaching 81% for the shared cache. Moreover, when the L2 is repartitioned according to our method, the relative variations of critical tasks execution times are less than 0.1% for the entire scenario switching frequency range we studied. We conclude that our dynamic cache repartitioning ensures reasonably high compositionality.

- We proved that our dynamic repartitioning increases application performance. In numbers, our scheme reduces the MPI of a shared cache with 33% on average, resulting in a 10% decrease of the average CPI. When compared with the static cache partitioning, on average, the superiority of the dynamic repartitioning is proven by the reduction in both MPI (19% on average, with a peak at 56%) and CPI (3% on average, with a peak at 10%).

In conclusion, we found that task centric cache management proved to be a potential approach for inducing compositionality for embedded multiprocessors systems with shared caches. In this dissertation we showed that both static and dynamic multimedia applications with soft real-time constraints may benefit from such a management scheme. The experimental results confirmed the idea that partitioning the cache per task basis induces cache compositionality. Moreover, they indicated that when assigning the cache parts according to the algorithms proposed in this thesis, the system performance is increased in most of the cases. In a nutshell, while preserving its flexibility, the shared cache acquired compositionality. In this manner the predictability of each application task is left undisturbed, and the analysis required to dimension the system can be performed in isolation for each task. Therefore the cost and effort involved in designing a system is reduced and reuse is facilitated, as detailed analysis of tasks in every possible combination is not required anymore. Ultimately,

these improvements lead to potentially cheaper platforms and shorter time to the market.

7.3 Future research directions

Our research may be continued in the following directions:

- In this dissertation we considered tasks as basic entities for cache management. But as a task may access large internal data arrays, further performance optimization may be possible by allocating parts of cache to the internal task's arrays (inside task compiler analysis may be combined with cache partitioning). In this case, a task would be able to allocate non-contiguous cache parts, resulting in new formulations for the cache allocation and cache mapping problems, and potential for more performance gain via cache partitioning.

- Minimizing energy consumption is important for a multiprocessor on chip in general, and for embedded multiprocessors that operate on battery power in particular. Thus, the trend in state of the art caches is to provide features for energy management. For example, cache lines that are not used may be turned to modes that save power in various degrees. Therefore, an interesting opportunity for both static and dynamic applications would be to use cache partitioning for power management.

- In this thesis, our main focus is on ensuring compositionality. When computing the partitioning ratio, our performance optimization methods do not take into account the task scheduling and mapping. Instead they target flexible systems, with natural load balancing of tasks to processors. Another option to compute the cache partitioning ratio is to take into account the scheduling policy, potentially leading to a higher cache utilization, hence better application performance, or cheaper platforms (if same application performance may be achieved with less L2 cache, the amount of cache on the platform can be decreased).

- Cache partitioning is a method that invalidates the potential benefits of cache sharing (namely more cache available per task), for the purpose of achieving compositionality. As we can notice, for applications with a large number of tasks, and platforms with small caches, partitioning the cache can cause performance degradation due to high cache fragmentation. For systems in which performance is an serious issue, and

having a small cache is a strong cost constraint, one can think of lowering the compositionality bounds for increasing performance. To do so, an idea is to share a cache partition among several tasks for which the maximum interference can be predicted. Another idea is to dynamically repartition the cache, even if the applications are static. In short, the compositionality-performance trade-off is a design parameter not yet explored.

- In the proposed dynamic repartitioning scheme we utilized a rather inaccurate estimate of the cache content reuse. The cache content reuse estimation can be refined, most likely leading to an increase in performance. Moreover, a fully dynamic repartitioning method that detects on-line the application's pattern of scenario switching and the cache content reuse, is likely to deliver further performance gain.

Overall, task centric cache management proved to bring interesting compositionality and performance advantages for multiprocessor platforms executing multimedia workload, and to open new exiting research directions.

Bibliography

[1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. In *SPAA '00: Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pages 1–12, New York, NY, USA, 2000. ACM.

[2] B. Akesson, K. Goossens, and M. Ringhofer. Predator: A Predictable SDRAM Memory Controller. In *Int'l Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 251–256. ACM Press New York, NY, USA, Sept. 2007.

[3] D. H. Albonesi. Selective cache ways: On-demand cache resource allocation. In *International Symposium on Microarchitecture*, pages 248–259, 1999.

[4] M. Alvarez, E. Salam, A. Ramirez, and M. Valero. A performance characterization of high definition digital video decoding using H.264/AVC. *IEEE International Symposium on Workload Characterization*, 2005.

[5] D. Andrade, B. B. Fraguela, and R. Doallo. Precise automatable analytical modeling of the cache behavior of codes with indirections. *ACM Trans. Archit. Code Optim.*, 4(3):16, 2007.

[6] F. Angiolini, L. Benini, and A. Caprara. Polynomial-time algorithm for on-chip scratchpad memory partitioning. *Proceedings, International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2003.

[7] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. pages 245–257, 2000.

[8] R. Banakar, S. Steinke, B. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. *Proceedings of the 10th International Workshop on Hardware/Software Codesign, CODES, Estes Park (Colorado)*, 2002.

[9] M. Bekooij, M. Wiggers, and J. van Meerbergen. Efficient buffer capacity and scheduler setting computation for soft real-time stream processing applications. pages 1–10, 2007.

[10] R. E. Bellman. *Dynamic Programming*. Dover Publications, Incorporated, 2003.

[11] G. E. Blelloch and P. B. Gibbons. Effectively sharing a cache among threads. *In Proceedings of SPAA*, pages 235–244, 2004.

[12] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal ACM*, 46(5):720–748, 1999.

[13] J. V. Busquets-Mataix and J. J. Serrano-Martin. The impact of extrinsic cache performance on predictability of real-time systems. In *RTCSA '95: Proceedings of the 2nd International Workshop on Real-Time Computing Systems and Applications*, page 8, 1995.

[14] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[15] Y. Cai, M. T. Schmitz, A. Ejlali, B. M. Al-Hashimi, and S. M. Reddy. Cache size selection for performance, energy and reliability of time-constrained systems. pages 923–928, 2006.

[16] S. Chakraborty, T. Mitra, A. Roychoudhury, L. Thiele, U. D. Bordoloi, and C. Derdiyok. Cache-aware timing analysis of streaming applications. In *ECRTS '07: Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 159–168, Washington, DC, USA, 2007. IEEE Computer Society.

[17] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. *In Proceedings of HPCA*, pages 340–351, 2005.

[18] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 242–252, New York, NY, USA, 2007. ACM.

[19] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. Scheduling threads for constructive cache sharing on cmps. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 105–115, New York, NY, USA, 2007. ACM.

[20] D. Chiou, P. Jain, L. Rudolph, and S. Devadas. Application-specific memory management for embedded systems using software-controlled caches. *Proceedings, Design Automation Conference*, 2000.

[21] D. T. Chiou. Extending the reach of microprocessors: Column and curious caching. *PhD thesis Department of EECS, MIT, Cambridge, MA*, 1999.

[22] L. Chunho, M. Potkonjak, and W. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. *Proceedings, International Symposium on Microarchitecture*, 1997.

[23] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[24] R. Cucchiara, M. Piccardi, and A. Prati. Exploiting cache in multimedia. *In International Conference on Computing and Systems, Florance, Italy*, 1999.

[25] E. A. de Kock and all. Yapi: application modeling for signal processing systems. *Proceedings, 37th conference on Design Automation*, pages 402 – 405, 2000.

[26] P. H. N. de With and E. G. Jaspers. On the design of multimedia software and future system architectures. *Embedded Processors for Multimedia and Communications*, 2004.

[27] A. Dominguez, S. Udayakumaran, and R. Barua. Heap data allocation to scratch-pad memory in embedded systems. *Journal of Embedded Computing*, 2005.

[28] H. Dybdahl and P. Stenstrom. "an adaptive shared/private nuca cache partitioning scheme for chip multiprocessors". *"In Proceedings of IEEE International Symposium of High Performance Computer Architecture"*, pages 2–12, 2007.

[29] M. K. Farrens, G. S. Tyson, and A. R. Pleszkun. A study of single-chip processor/cache organizations for large numbers of transistors. In *ISCA*, pages 338–347, 1994.

[30] M. Frigo and V. Strumpen. The cache complexity of multithreaded cache oblivious algorithms. In *SPAA '06: Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 271–280, New York, NY, USA, 2006. ACM.

[31] J. Fritts. Multi-level memory prefetching for media and stream processing. *Proceedings, International Conference on Multimedia and Expo*, pages 2–13, 2002.

[32] J. Fritts, W. Wolf, and B. Liu. Understanding multimedia application characteristics for designing programmable media processors. *In Media Processors, volume 3655 of Proceedings of SPIE*, pages 2–13, 1999.

[33] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[34] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: an analytical representation of cache misses. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pages 317–324, 1997.

[35] K. Goossens and M. Bekooi. "private conversation". 2007.

[36] A. Gordon-Ross, F. Vahid, and N. Dutt. Fast configurable-cache tuning with a unified second-level cache. pages 323–326, 2005.

[37] A. Hartstein, V. Srinivasan, T. R. Puzak, and P. G. Emma. Cache miss behavior: is it sqrt(2)? pages 313–320, 2006.

[38] J. L. Hennesy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, San Fransisco, CA, 2003.

[39] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni. Communist, utilitarian, and capitalist cache policies on cmps: caches as a shared resource. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 13–22, New York, NY, USA, 2006. ACM.

[40] IBM. IBM PowerPC 970 Data Sheet. http://www-306.ibm.com.

[41] IMEC. IMEC reconfigurable systems program. http://www.imec.be/ovinter/static_research/reconfigurable.shtml.

[42] Intel. Intel Core 2 Extreme Quad-Core Processor Q6000 Datasheet.

[43] J. Irwin, D. May, H. Muller, and D. Page. Predictable instruction caching for media processors. *13th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 141–150, 2002.

[44] I. Issenin, E. Brockmeyer, and M. Miranda. Data reuse analysis technique for software-controlled memory hierarchies. *Proceedings, Design, Automation and Test in Europe*, 2004.

[45] ITRS. International Technology Roadmap for Semiconductors-Executive Summary, 2007. http://www.itrs.net/Common/2007ITRS/ExecSum2007.pdf.

[46] ITU. H.264 Video Coding Standard, 2008. http://www.itu.int/rec/T-REC-H.264/en.

[47] R. Iyer. Cqos: a framework for enabling qos in shared caches of cmp platforms. *In Proceedings of the 18th Annual International Conference on Supercomputing*, pages 257–266, 2004.

[48] R. E. Jan Staschulat. Multiple process execution in cache related preemption delay analysis. *Proceedings, ACM International Conference on Embedded Software (EMSOFT)*, pages 278–286, 2004.

[49] A. Janapsatya, S. Parameswaran, and A. Ignjatovic. Hardware/software managed scratchpad memory, for embedded systems. *Proceedings, International Conference on Computer Aided Design*, 2004.

[50] E. G. Jaspers, R. Gelderblom, and M. Tomas. Padme: Multiprocessor h.264 decoding. 2003.

[51] I. Kadayif, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Hardware-software co-adaptation for data-intensive embedded applications. 2002.

[52] A. Kalavade and P. Moghé. A tool for performance estimation of networked embedded end-systems. In *DAC '98: Proceedings of the 35th annual conference on Design automation*, pages 257–262, New York, NY, USA, 1998. ACM.

[53] M. Kandemir and A. Choudhary. Compiler-directed scratchpad memory hierarchy design and management. *Proceedings, 39th Design Automation Conference*, 2002.

[54] M. Kandemir, J. Ramanujam, and A. Choudhary. Exploiting shared scratch pad memory space in embedded multiprocessor systems. *Proceedings, 39th Design Automation Conference*, 2002.

[55] U. Kellerer, H. Pferschy and D. Pisinger. *Knapsack Problems*. Springer Verlag, 2005.

[56] C. Kim, S. Chung, and C. Jhon. An energy-efficient partitioned instruction cache architecture for embedded processors. *IEICE - Trans. Inf. Syst.*, E89-D(4):1450–1458, 2006.

[57] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. *In Proceedings of IEEE PACT*, pages 111–122, 2004.

[58] D. B. Kirk. Smart (strategic memory allocation for real-time) cache design. *IEEE symposium on Real Time Systems*, pages 229–237, 1989.

[59] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983.

[60] R. Koenen. Overview of the mpeg-4 standard. 2002.

[61] S. Kohli. *Cache aware scheduling for synchronous dataflow programs*. Master Thesis, University of California, Berkeley, 2004.

[62] C.-G. Lee, K. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C.-S. Kim. Bounding cache-related preemption delay for real-time systems. *IEEE Trans. Software Eng.*, 27(9):805–826, 2001.

[63] J. Liedtke, H. Härtig, and M. Hohmuth. Os-controlled cache predictability for real-time systems. *3rd IEEE Real-Time Technology and Applications Symposium*, June 1997.

[64] J. W. S. W. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[65] MIPS. MIPS32 Architecture. http://www.mips.com/content/ Documentation/MIPSDocumentation/ProcessorArchitecture/doclibrary.

[66] A. Molnos, S. Cotofana, M. Heijligers, and J. van Eijndhoven. Static cache partitioning robustness analysis for embedded on-chip multiprocessors. *In Proceedings of the ACM International Conference on Computing Frontiers*, pages 353–360, 2006.

[67] A. Molnos, S. D. Cotofana, M. Heijligers, and J. Eijndhoven. Static cache partitioning robustness analysis for embedded on-chip multiprocessors. *Transactions on High-Performance Embedded Architectures and Compilers I*, pages 269–288, August 2007.

[68] A. Molnos, S. D. Cotofana, M. Heijligers, and J. Eijndhoven. Compositional, dynamic cache management for embedded chip multiprocessors. *Journal of VLSI Signal Processing Systems, Special Issue on Multicore Enabled Multimedia Applications & Architectures*, 2008.

[69] A. Molnos, S. D. Cotofana, M. J. M. Heijligers, and J. T. J. van Eijndhoven. Throughput optimization via cache partitioning for embedded multiprocessors. In *Proceedings of 2006 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS 2006), Samos, Greece, July 17-20, 2006*, pages 185–192, 2006.

[70] A. Molnos, M. Heijligers, S. Cotofana, and J. van Eijndhoven. Compositional memory systems for data intensive applications. *Proceedings, Design, Automation and Test in Europe*, pages 728–729, 2004.

[71] A. Molnos, M. Heijligers, S. Cotofana, and J. van Eijndhoven. Compositional memory systems for multimedia communicating tasks. *Proceedings, DATE*, 2005.

[72] A. Molnos, M. Heijligers, S. Cotofana, and J. van Eijndhoven. Compositional, efficient caches for a chip multi-processor. *Proceedings, Design, Automation and Test in Europe*, pages 345–350, 2006.

[73] A. Molnos, M. J. M. Heijligers, and S. D. Cotofana. Compositional, dynamic cache management for embedded chip multiprocessors. In *Proceedings, Design, Automation and Test in Europe, DATE 2008, Munich, Germany, March 10-14, 2008*, pages 991–996, 2008.

[74] G. Moore. Cramming more components on integrated circuits. *Electronics, April 19*, 1965.

[75] T. C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD Thesis, Standford University, 1994.

[76] MPEG.ORG. The reference website for the MPEG standard. http://www.mpeg.org.

[77] F. Mueller. Compiler support for software-based cache partitioning. *ACM SIGPLAN Notices*, 30(11), 1995.

[78] H. Muller, D. Page, J. Irwin, and D. May. Caches with compositional performance. *Proceedings, Embedded Processor Design Challenges*, pages 242–259, 2002.

[79] A. C. Nacul and T. Givargis. Dynamic voltage and cache reconfiguration for low power. 2004.

[80] B. A. Nayfeh and K. Olukotun. Exploring the design space for a shared-cache multiprocessor. pages 166–175, 1994.

[81] H. S. Negi, T. Mitra, and A. Roychoudhury. Accurate estimation of cache-related preemption delay. *Proceedings, CODES+ISSS*, pages 201–206, 2003.

[82] NXP. Trimedia programmable media processor databook. http://www.nxp.com.

[83] P. R. Panda, N. D. Dutt, and A. Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. *Proceedings, European Design and Test Conference*, 1997.

[84] P. Petoumenos, G. Keramidas, H. Zeffer, S. Kaxiras, and E. Hagersten. Modeling cache sharing on chip multiprocessor architectures. In *IISWC*, pages 160–171, 2006.

[85] D. Pisinger. *Algorithms for Knapsack Problems*. PhD thesis, University of Copenhagen, Dept. of Computer Science, feb 1995.

[86] M. M. Planas, F. Cazorla, A. Ramirez, and M. Valero. Explaining dynamic cache partitioning speed ups. volume 6, Washington, DC, USA, 2007. IEEE Computer Society.

[87] G. Pokam and F. Bodin. Energy-efficiency potential of a phase-based cache resizing scheme for embedded systems. pages 53–62, 2004.

[88] A. Prati. Exploring multimedia applications locality to improve cache performance. In *MULTIMEDIA '00: Proceedings of the eighth ACM international conference on Multimedia*, pages 509–510, 2000.

[89] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–432, Washington, DC, USA, 2006. IEEE Computer Society.

[90] M. K. Qureshi, D. Thompson, and Y. N. Patt. The v-way cache : Demand-based associativity via global replacement. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 544–555, Washington, DC, USA, 2005. IEEE Computer Society.

[91] H. Ramaprasad and F. Mueller. Bounding worst-case data cache behavior by analytically deriving cache reference patterns. In *RTAS '05: Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, pages 148–157, 2005.

[92] P. Ranganathan, S. Adve, and N. P. Jouppi. Reconfigurable caches and their application to media processing. 2000.

[93] V. Reyes, W. Kruijtzer, T. Bautista, G. Alkadi, and A. Nunez. A unified system-level modeling and simulation environment for MPSoC design: MPEG-4 decoder case study. *Proceedings, Design, Automation and Test in Europe*, 2006.

[94] I. Richardson. *H.264 and MPEG-4 Video Compression Video Coding for Next-Generation Multimedia: Video Coding for Next-generation Multimedia*. John Wiley and Sons Ltd., 2003.

[95] F. Sebek. The state of the art in cache memories and real-time systems. Technical Report 01/37, Mälardalen Real-Time Research Centre, Sweden, Department of Computer Engineering, Mälardalen University, Sweden, oct 2001.

[96] J. Sermulins, W. Thies, R. Rabbah, and S. Amarasinghe. Cache aware optimization of stream programs. *SIGPLAN Not.*, 40(7):115–126, 2005.

[97] A. Settle, D. Connors, E. Gibert, and A. Gonzlez. "a dynamically reconfigurable cache for multithreaded processors". *"in Journal of Embedded Computing "*, pages 221 – 233, 2006.

[98] M. Shalan and V. J. Mooney. A dynamic memory management unit for embedded real-time system-on-a-chip. *Proceedings of the 2000 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 180–186, 2000.

[99] N. T. Slingerland and A. J. Smith. Cache performance for multimedia applications. *In Thirteenth IASTED International Conference on Parallel and Distributed Computing System*, pages 18–21, June 2001.

[100] S. Srikantaiah, M. Kandemir, and M. J. Irwin. Adaptive set pinning: managing shared caches in chip multiprocessors. *SIGARCH Comput. Archit. News*, 36(1):135–144, 2008.

[101] P. Stenstrom. A survey of cache coherence schemes for multiprocessors. *IEEE Computer*, pages 12–24, 1990.

[102] A. Stevens. Level 2 cache for high-performance arm core-based soc systems. *ARM white paper*, 2004.

[103] H. S. Stone, J. Truek, and L. Wolf, Joel. Optimal partitioning of cache memory. *IEEE Transactions on computers*, 41(9):1054–1068, 1992.

[104] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, 2004.

[105] SystemC. Language Reference Manual for SystemC 2.1, 2005. http://www.systemc.org.

[106] Y. Tan and V. J. Mooney. A prioritized cache for multi-tasking real-time systems. pages 168–175, 2003.

[107] Y. Tan and V. J. Mooney. Integrated intra- and inter-task cache analysis for preemptive multi-tasking real-time systems. *Proceedings, SCOPES*, pages 182–199, 2004.

[108] A. S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 2001.

[109] A. Terechko. Hardware cache coherence prototyping for the tm2270 trimedia. *Philips Research Technical Note PR-TN 2005/00312*, 2005.

[110] TI. TMS320DM6443 Product Folder. http://www.ti.com.

[111] H. Tomiyama and N. D. Dutt. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. *CODES*, pages 67–71, 2000.

[112] S. Udayakumaran and R. Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. *Proceedings, International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2003.

[113] P. van der Wolf, E. de Kock, T. Henriksson, W. Kruijtzer, and G. Essink. Design and programming of embedded multiprocessors: an interface-centric approach. pages 206–217, 2004.

[114] J. T. van Eijndhoven, J. Hoogerbrugge, M. Jayram, P. Stravers, and A. Terechko. *Chapter: Cache-Coherent Heterogeneous Multiprocessing as Basis for Streaming Applications, in Dynamic and robust streaming between connected CE-devices.* Kluwer Academic Publishers, 2005.

[115] J. T. van Eijndhoven, J. Hoogerbrugge, M. Jayram, P. Stravers, and A. Terechko. *Dynamic and robust streaming between connected CE-devices.* Kluwer Academic Publishers, 2005.

[116] S. P. Vanderwiel and D. J. Lilja. Data prefetch mechanisms. *ACM Comput. Surv.*, 32(2):174–199, 2000.

[117] M. H. Wiggers, M. J. G. Bekooij, P. G. Jansen, and G. J. M. Smit. Efficient computation of buffer capacities for cyclo-static real-time systems with back-pressure. pages 281–292, 2007.

[118] C. Zhang, F. Vahid, and R. Lysecky. A self-tuning cache architecture for embedded systems. *Trans. on Embedded Computing Sys.*, 3(2):407–425, 2004.

[119] Y. Zhong, S. G. Dropsho, and C. Ding. Miss rate prediction across all program inputs. In *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 79–91, 2003.

List of Publications

Journal Papers

1. **A.M. Molnos**, S. D. Cotofana, M.J.M. Heijligers, J.T.J. Eijndhoven, "*"Compositional, dynamic cache management for embedded chip multiprocessors"*", Journal of VLSI Signal Processing Systems, Special Issue on Multicore Enabled Multimedia Applications & Architectures, 2008.

2. **A.M. Molnos**, S. D. Cotofana, M.J.M. Heijligers, J.T.J. Eijndhoven, "*Static Cache Partitioning Robustness Analysis for Embedded On-Chip Multi-processors*", Transactions on High-Performance Embedded Architectures and Compilers I, pp. 279-297, August 2007.

International Conferences

1. **A.M. Molnos**, M.J.M. Heijligers, S.D. Cotofana, J.T.J. Eijndhoven, "*Compositional, dynamic cache management for embedded chip multiprocessors*", Proceedings of the Design, Automation and Test in Europe, (DATE'08), 2008.

2. **A.M. Molnos**, M.J.M. Heijligers, S.D. Cotofana, J.T.J. van Eijndhoven, "*Compositional, efficient caches for a chip multi-processor*", Proceedings of the Design, Automation and Test in Europe, (DATE'06), 2006.

3. **A.M. Molnos**, , S.D. Cotofana, M.J.M. Heijligers, J.T.J. van Eijndhoven, "*Throughput optimization via cache partitioning for embedded multiprocessors*", Proceedings of the IEEE International Conference on Embedded Computer Systems: Architectures, MOdeling, and Simulation, 2006.

4. **A.M. Molnos**, S.D. Cotofana, M.J.M. Heijligers, J.T.J. van Eijndhoven, "*Static cache partitioning robustness analysis for embedded on-chip multi-processors*", Proceedings of the ACM International Conference on Computing Frontiers, 2006.

5. **A.M. Molnos**, M.J.M. Heijligers, S.D. Cotofana, J.T.J. van Eijndhoven, "*Compositional memory systems for multimedia communicating tasks*", Proceedings of the Design, Automation and Test in Europe, (DATE'05), 2005.

6. **A.M. Molnos**, M.J.M. Heijligers, S.D. Cotofana, J.T.J. van Eijndhoven, "*Compositional memory systems for data intensive applications*", Proceedings of the Design, Automation and Test in Europe 2004 (DATE'04), 2004.

Local Conferences

1. **A.M. Molnos**, M.J.M. Heijligers, S. D. Cotofana, J.T.J. Eijndhoven , "*Inter-task sharing data and instructions in cache with enabling compositionality in parallel embedded systems*", Proceedings of the 16th Annual Workshop on Circuits, Systems and Signal Processing, ProRISC 2005.

2. **A.M. Molnos**, M.J.M. Heijligers, S.D. Cotofana, J.T.J. Eijndhoven, "*Cache Partitioning Options for Compositional Multimedia Applications*" Proceedings of the 5th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC) 2004

3. **A.M. Molnos**, M.J.M. Heijligers, S.D. Cotofana, J.T.J. van Eijndhoven, B. Mesman, "*Data Cache Optimisations in Multimedia Applications*", Proceedings of the 14th Annual Workshop on Circuits, Systems and Signal Processing, ProRISC 2003

Publications not directly related to this thesis

1. R. Dobrescu, D. Andone, **A.M. Molnos**, M. Guzu, E. Vitos, "*Server Architecture for UNI Signalling over ATM Networks*", Proceedings of the IEEE International Conference on Telecommunication (ICT2001), 2001

Samenvatting

In deze dissertatie stellen wij een cache geheugen management methode voor die toegepast kan worden in embedded chips met meerdere processoren die multimedia applicaties uitvoeren met soft-real time eisen. Hierbij nemen wij een CAKE multiprocessor platform met 4 TriMedia processoren en een interne geheugenhiërarchie met een gedeelde level twee (L2) cache als uitgangspunt. Omdat de interferentie normaal gesproken onvoorspelbaar is indien er gezamenlijke toegang is naar de gedeelde L2 cache, dient het systeem compositioneel te zijn om aan zijn real-time eisen te kunnen voldoen. Met andere woorden, de prestaties van elk individueel deel (of taak) dienen behouden te blijven wanneer meerdere verschillende taken tegelijkertijd uitgevoerd worden of indien er taken worden toegevoegd. Om compositionele cache toegang te garanderen stellen wij voor om delen van de L2 exclusief toe te wijzen aan elke taak. Wij vergelijken vervolgens set- en associativiteits-gebaseerde partitionering van de L2 voor applicaties bestaande uit onafhankelijke taken. We gebruiken het aantal cache misses dat veroorzaakt wordt door inter-taak cache interferentie als een maat van compositionaliteit. We zien dat beide partitioneringen een grote mate van compositionaliteit opleveren (het aantal cache misses dat veroorzaakt wordt door inter-taak cache interferentie is kleiner dan 1% van het aantal cache misses voor de gehele applicatie). Set-gebaseerde partitionering verhoogt, en associativiteits-gebaseerde partitionering verlaagt de prestaties van het systeem. Dit is de reden waarom we set-gebaseerde partitionering hebben gekozen als basis voor onze taak-centrische cache management methode. Voor applicaties met afhankelijke taken introduceren wij een gemengde partitioneringsmethode. Deze methode wijst de cache set-gebaseerd toe aan elke taak en elk gedeeld data/code bereik, en wijst de cache vervolgens associativiteits-gebaseerd toe binnen elk gedeeld data/code bereik aan iedere taak die dit bereik addresseert. Vervolgens stelden wij optimalisatie strategieën van de cache partitionerings verhouding voor om het aantal cache misses te minimaliseren en om de doorstroom te maximaliseren. Experimenten tonen

aan dat deze gemengde partitioneringsmethode resulteert in minder dan 1% L2 interferentie tussen de taken, en dus een hoge compositionaliteit tot gevolg heeft, en tevens de prestaties verbetert. Tenslotte stellen wij, voor scenario's waarin applicatie taken kunnen starten of stoppen, een dynamische cache herpartitioneringsmethode voor die: (1) in de ontwerpfase de locatie in de cache voor iedere taak in ieder scenario bepaalt, waarbij de kritieke taken niet verstoord worden en de herpartitioneringskosten minimaal zijn, en (2) tijdens de uitvoering schakelt tussen de verschillende partitioneringen en de repartitioneringskosten nog verder verlaagt. Simulaties suggereren dat deze de herpartitioneringsmethode een hoge mate van compositionaliteit oplevert, dat kritieke taken gewaarborgd blijven, en dat de prestatie wordt verhoogd. De verkregen resultaten geven aan dat taak-centrisch cache management een veelbelovende aanpak is voor embedded multiprocessor systemen die statische of dynamische multimedia applicaties uitvoeren.

Curriculum Vitae

Anca Mariana Molnoş was born in Făgăraş, Romania on the 10^{th} of December 1977. After finishing her the secondary education at the "Radu Negru" secondary school in Făgăraş, from 1996 she was a student of the Faculty of Automatic Control and Computers Science at the "Politehnica" University of Bucharest. There she obtained the *Computer Engineer* degree in 2001. During one semester (in 2001-2002), she was a teaching assistant at the Faculty of Automatic Control and Computers Science, "Politehnica" University of Bucharest.

In 2002 she joined as a Ph.D. student the Computer Engineering Group at Delft University of Technology, where she performed research on cache management for embedded multiprocessors, under the supervision of Prof.Dr. Stamatis Vassiliadis and Dr. Sorin Coţofana. Her Ph.D. activity was supported by a doctoral fellowship from Philips Research Laboratories in Eindhoven, where she was a guest scientist under the supervision of Dr. Marc Heijligers.

Her main research interests include multiprocessor architecture, memory management and low-power techniques for multiprocessors, and embedded systems in general.