J Sign Process Syst
DOI 10.1007/s11265-008-0276-5

Compositional, Dynamic Cache Management for Embedded Chip Multiprocessors

Anca M. Molnos · Sorin D. Cotofana · Marc J. M. Heijligers · Jos T. J. van Eijndhoven

Received: 14 September 2007 / Revised: 23 June 2008 / Accepted: 2 September 2008

Abstract This paper proposes a dynamic cache repartitioning technique that enhances compositionality on platforms executing media applications with multiple utilization scenarios. Because the repartitioning between scenarios requires a cache flush, two undesired effects may occur: (1) in particular, the execution of critical tasks may be disturbed and (2) in general, a performance penalty is involved. To cope with these effects we propose a method which: (1) determines, at design time, the cache footprint of each tasks, such that it creates the premises for critical tasks safety, and minimum flush in general, and (2) enforces, at run-time, the design time determined cache footprints and further decreases the flush penalty. We implement our dynamic cache management strategy on a CAKE multiprocessor with 4 Trimedia cores. The experimental workload consists of 6 multimedia applications, each of which formed by multiple tasks belonging to an extended MediaBench suite. We found on average that: (1) the relative variations of critical tasks execution time are less than 0.1%, regardless of the scenario switching frequency, (2) for realistic scenario switching frequencies the inter-task cache interference is at most 4% for the repartitioned cache, whereas for the shared cache it reaches 68%, and (3) the off-chip memory traffic reduces with 60%, and the performance (in cycles per instruction) enhances with 10%, when compared with the shared cache.

A. M. Molnos (✉) · M. J. M. Heijligers
NXP Semiconductors, HTC 37, Eindhoven, The Netherlands
e-mail: anca.molnos@nxp.com

A. M. Molnos · S. D. Cotofana
Technical University of Delft, Mekelweg 4,
Delft, The Netherlands

J. T. J. van Eijndhoven
Vector Fabrics, Kanaaldijk Zuid 15,
Eindhoven, The Netherlands

Keywords Multiprocessor · Cache management · Compositionality · Predictability

1 Introduction

Over the last years, the size and complexity of multimedia applications have a clear tendency to increase. As a result, such applications demand more and more performance from the underlying hardware platforms. In the embedded field a common practice to boost performance is to use multiprocessor architectures. Moreover, we assist to a sustained progress of the semiconductor technology, hence multiple processors can be integrated on a single chip, forming a so-called Chip Multi-Processor (CMP). Nevertheless the speed gap between the processors and the off-chip memory widens with 50% every year [17]. Therefore, to mitigate this gap, a CMP typically comprises a number of memory buffers.

Cache hierarchies [7] represent a possible organization of the on-chip memory buffers. In this paper we consider a CMP with a memory hierarchy in which each processor core has its own level one (L1) cache, and the platform has a large level two (L2) cache shared among all the cores [20, 26]. Furthermore, we assume that a multi-processor executes a software application (referred in this paper shortly as "application") consisting of a given task set. As the present work is in the embedded system area, all the application tasks are known at design time. When used in conjunction with a CMP architecture, shared caches make the miss rate prediction difficult because different tasks executed in parallel may flush each other data at random. Unpredictability constitutes a major problem for media applications for which the completion of tasks before their deadlines is of crucial importance (for instance a video decoder has to compute 25 frames in a second). Ideally, the designer would like to be able to predict the overall application performance based on the performance of its individual tasks. In particular, the performance of each task must be preserved if the tasks are executed concurrently in arbitrary combinations or if additional tasks are added. A system satisfying this property is addressed as having *compositional* performance.

Cache partitioning among tasks was proposed in order to diminish the inter-tasks interference in cache [11, 15, 18, 19, 21, 24]. These articles target applications composed out of tasks that all execute for the entire lifetime of the application. However, a typical multimedia application may have multiple utilization scenarios, in the sense 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. Therefore, cache management strategies have to be able to deal with such dynamic applications, while preserving compositionality.

In this paper we propose a strategy to dynamically repartition the cache at a scenario change, such that the compositionality is enabled. This strategy is based on determining the best static partition for each possible utilization 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 tasks 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) 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 that the data flush is actually not needed anymore.

In the envisaged architecture the L2 is shared among the processors, thus it is heavily affected by inter-task conflicts. Consequently, the cache management method targets the L2. We exercise the repartitioning approach on a CAKE platform [26] with 4 Trimedia cores executing 6 multimedia parallel applications, and we investigate a wide scenario switching frequency range (from 100 Hz to 1 Hz). We found that for realistic scenario switching frequencies above 10 Hz the inter-task cache interference is at most 4% for the repartitioned cache, whereas for the shared cache it reaches 68%, indicating that the proposed strategy achieves high compositionality. Moreover, the relative variations in critical tasks execution times are less than 0.1%, for all the studied scenario switching frequencies, suggesting that the critical tasks remain un-disrupted. In addition, on average, the dynamic repartitioning method reduces the off-chip memory traffic (measured in L2 misses per instruction) with 60%, when compared with the shared cache and with 25% when compared with a statically partitioned cache. As a consequence, the average number of cycles needed to execute an instruction is decreased with 10%, when compared with the shared cache, and with 4% when compared with a statically partitioned cache.

This paper is organized as follows. Section 2 introduces the considered multiprocessor architecture, the possible cache partitioning types, and discusses existing work in this domain. This is followed by a description of the proposed cache repartitioning method in Section 3. The experimental results are presented in Section 4. The related work is discussed in Section 5 and finally Section 6 concludes the paper.

2 Background

The envisaged multi-processor architecture consists of a multi-processor like the one presented in Fig. 1, comprising several media processors and a control processor. These processors are connected to on-chip memory banks by a fast, high-bandwidth interconnection network.

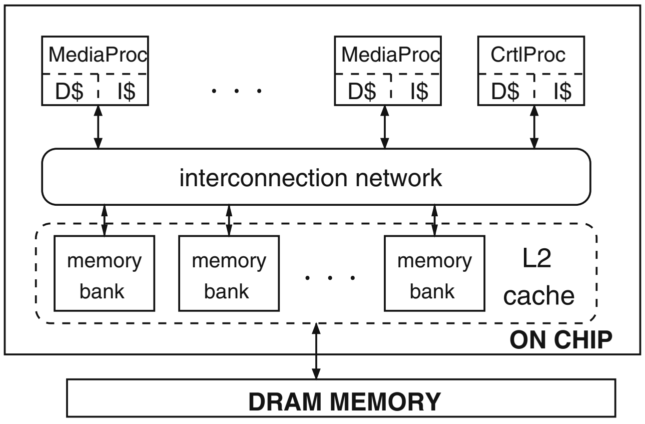


Figure 1 Multi-processor target architecture.

The memory hierarchy is organized as follows: on the first level there are the L1 caches private to each processor core, on the next level it is an on chip L2 shared by all processors, and on the last level in the hierarchy it is an off-chip main memory. The L1 caches are split among instructions and data, and the L2 cache is unified. Among the L1s and the L2 a hardware protocol is implemented to ensure coherence. The L1s and the L2 use a write back policy in order to decrease the accesses among the memory hierarchy levels.

In general, an application A executed on this architecture consists of a task set $\mathcal{T} = \{T_i\}_{(i=1,2,\ldots,N)}$ and has a set $\mathcal{S} = \{S_q\}_{(q=1,2,\ldots,Z)}$ of possible scenarios. In each scenario S_q only a subset of tasks $\mathcal{T}_q \subseteq \mathcal{T}$ is active. In this paper we consider the case of soft real time applications. However, some task may be less tolerant to disturbance than others. Let us look for example at a device able to record a video stream and play another stream at the same time. A short stall of the video decoder might result in omitting to display a frame, which may be a reasonable quality loss. On the contrary, a short stall in the recording task may result in a large quality loss, depending on which stream part the device failed to record. We denote such tasks that cannot tolerate disturbances as critical. Note that critical tasks do not have to be active in all application scenarios. The scenarios and the critical tasks specification has to be performed by the application designer, and it is not the subject of this paper.

On the targeted platform, in a given scenario, multiple tasks may execute concurrently possibly accessing the L2. If no precautions are taken, for instance, when task T_i's data are loaded into the cache, they may flush task T_j's data, eventually causing a future T_j miss. In this manner the system is not compositional and the predictability cannot be guaranteed. Our work targets this L2 cache contention, therefore we focus on isolating tasks such that their number of misses are independent of each other, even though the scenarios may change. We assume that the L1s are not subject to the aforementioned inter-task cache contention. This is a reasonable assumption, as an L1 is private to each task during its execution. We further assume that other shared resources like buses, communication networks, etc. are managed for compositionality using methods like the ones described in [27].

An existing manner to induce compositionality is to assign to each task an exclusive cache part. In the organization of a conventional, set associative cache the address splits in three parts: tag, index, and offset [7]. The index directly addresses a cache set (row). Every set has a number of M ways (column). The tag part of the address is compared against all the tag parts stored in a set to determine if there is a hit in one of the set's ways. The offset part of the address selects the desired word in the cache block. With respect to conventional cache organization we identify two possible types of partitioning:

– Associativity based, also called column caching [3] (Fig. 2a). In this situation a task gets a number of ways from every set of the cache.
– Set based (Fig. 2b). In this situation a task gets a number of sets from the cache.

The associativity based partitioning is mostly used in the literature [23, 24] because its implementation requires only a small change in the cache replacement policy. However, in the context of compositionality, the main shortcoming of associativity based approaches 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, while in media applications there is a trend in adding new features, so in increasing the number of tasks. For such an application there might be not enough ways for every task, therefore multiple tasks would share the same way, leading to unforeseeable cache interference, hence to an un-compositional system. Moreover, it is known from the literature [7] that if a program may use only few ways, the cache efficiency degrades. In [14] a brief quantitative comparison among the static set and associative cache partitioning is presented. An in depth comparison is included in [13]. As expected, the conclusion of this comparison is that the associativity based partitioning achieves compositionality, but degrades the cache performance.

The set based partitioning is more difficult to implement as all the addresses of a task have to map exclusively in a restricted cache region, allocated to that task. However, due to the fact that typically in a cache there are thousands of sets, the set based partitioning can potentially induce compositionality, therefore this is the partitioning type we consider in this paper. In the

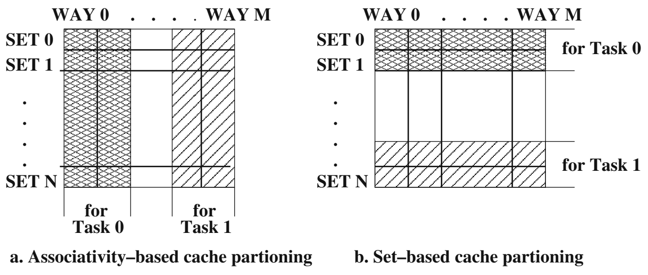


Figure 2 Types of cache partitioning (**a**, **b**).

following we describe the chosen implementation for set based partitioning.

We achieve the cache partitioning through a level of indirection, without interfering with the memory space. Our scheme modifies the index bits of an address into new index bits, before cache lookup (Fig. 3), taking into account who initiates the access. The purpose of the index translation is to send all 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 a power of two number of sets. We propose to use a table (indexed by the *task id*) that provides the information needed for the index translation ($MASK$ and $BASE$ bits). To clarify the mechanism, let us assume that an access to data A belonging to task T_i has the index idx_A, in a conventional cache case. 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 considered in number of sets). The $MASK_A$ bits actually select the k least significant bits of idx_A (instead of doing modulo with the cache size 2^C we do modulo with the partition size 2^k). $BASE_A$ 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 do not have the same old index might end up having the same new index. In this case the system is not able to distinguish between 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 significant 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

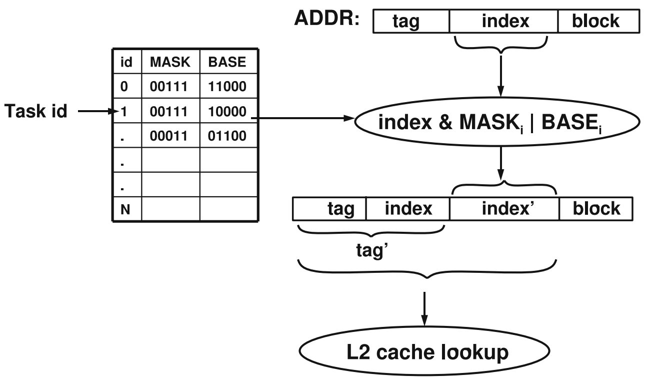


Figure 3 Set based cache partitioning (implementation).

size) we choose to augment the tag with all index bits. In this case, for instance, for a 2 MBytes L2, 8 ways associative, 512Bytes block size, the tag has 9 extra bits, representing less that 0.5% of the total L2 area, so the implied area penalty can be considered negligible.

In this work we assume a multiprocessor platform with cache coherence among the L1 caches of each processor core, as previously mentioned. 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 [25] 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. In the case of an L2 miss, the cache is refilled with a new data block from memory, thus the shadow tags have to be synchronized with the tag of the data newly brought in the L2. However this synchronization is not on the critical path, because it is performed while waiting for the data refill from the memory.

3 Dynamic, Set Based, Cache Repartitioning

We start this section by first introducing some useful notations. We consider that in scenario S_q the cache size of a task $T_i \in \mathcal{T}_q$ is denoted with $c_{i,q}$. 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 resides, $cf_{i,q} = [b_{i,q}, b_{i,q} + c_{i,q})$, where $b_{i,q} \in [1, C - c_{i,q}]$ 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 $T_i \in \mathcal{T}_q$.

Cache partitioning is designed to isolate the tasks in the cache therefore to enhance 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 *cache allocation problem*, $\mathcal{CAP}$ (find the cache sizes c_i);
2. the *cache mapping problem*, $\mathcal{CMP}$ (find the cache footprints cf_i).

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 in the existing static partitioning methods the problem of interest is the *cache allocation*. However, at a scenario transition $S_q \rightarrow S_w$, the repartitioning costs may depend on $cf_{i,q}$ and $cf_{i,w}$. In the case of set based partitioning, if T_i's footprints $cf_{i,q}$ and $cf_{i,w}$ are completely disjoint, $cf_{i,q} \cap cf_{i,w} = \{\varnothing\}$, precautions should be taken such that, after repartitioning, T_i accesses the most recent data copy. Eventually, T_i data have to be relocated into the new T_i's cache part at the scenario transition via flushing or other strategy that typically involves a form of overhead. On the contrary, the ideal case occurs when the cache footprint of T_i is the same in both scenarios ($b_{i,q} = b_{i,w}$ and $c_{i,q} = c_{i,w}$), thus no relocation overhead is present. In conclusion, in dynamic repartitioning the performance of the systems heavily relies on $cf_{i,q}$ and $cf_{i,w}$, therefore the *cache mapping problem* becomes interesting.

This paper presents a dynamic cache management strategy consisting of two parts. We propose a method to solve the cache mapping problem at design time. Already at this stage the method creates the premises for guaranteed non disturbance of critical task and a minimal cache repartitioning penalty. For run time we introduce a cache controller extension able to impose the statically determined footprints and to further decrease the penalty involved in cache flushing. In the next subsections we first detail the implications of set based cache repartitioning, after which we propose an heuristic to statically determine the cache footprints, and in the last part of this section we present the run time cache management strategy.

3.1 Set Based Cache Repartitioning

In this subsection we investigate the cases when the cache content can be reused, at a scenario change. We would like to mention that we do not assume the existence of a possibility to directly transfer data from one L2 set to another, 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 required. Such mechanisms are in principle possible but in order to minimize the hardware overhead we do not embed them in our current proposal. Thus for now, our option is to flush the T_i's footprint corresponding to the old scenario S_q. Later, 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.

Due to implementation reasons, the number of L2 sets a task can own is a power of two. For simplicity sake, let us assume that for a scenario switch both cache footprints begin at 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 Fig. 4. 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 stays in the same location in the double sized footprint $cf_{i,w}$. Therefore, to keep data correctness, one has to flush only the data that does not map anymore in $cf_{i,w}$ in S_w. However, in order to determine which data fall into this category a search similar to the conventional cache look-up should be performed on the cache lines at a scenario change. We do not assume the existence of such a mechanism, 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 Fig. 5, 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 Fig. 5) are relocated in $cf_{i,w}$, when the scenario becomes S_w. In conclusion, in order to keep data correctness, only 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

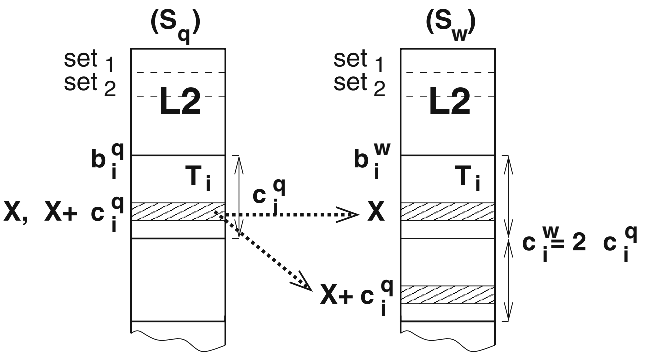


Figure 4 Cache repartitioning—doubling the size.

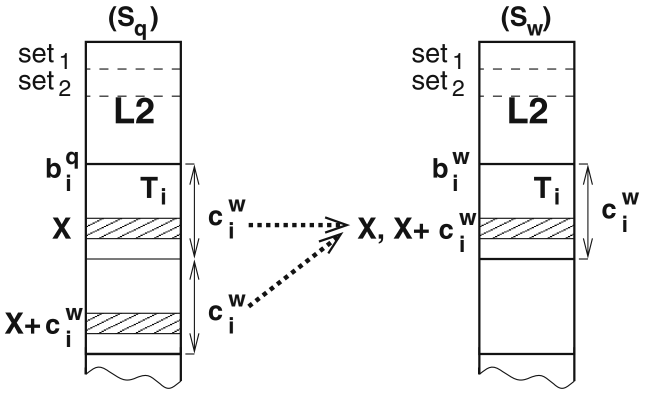


Figure 5 Cache repartitioning—halving the size.

$S_q \to S_w$ 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 $\varkappa \in \mathbb{N}$, $\varkappa < c_{i,q}/c_{i,w}$.

The footprint set of a critical task is denoted as "sane" if a complete cache content reuse is guaranteed at each possible scenario change. Taking into account the considerations above, one can see that the footprint of a critical tasks T_i is sane if $\forall q, \forall w, cf_{i,q} = cf_{i,w}, T_i \in \mathcal{T}_q, T_i \in \mathcal{T}_w$ (critical task's data should be cached always in the same place).

3.2 Cache Content Reuse via Footprint Management

In order to solve the cache mapping problem, we need to know the cache sizes allocated to each task in each scenario. To determine these cache sizes we use the method proposed in [15] that minimizes the total application number of misses. Moreover, we assume that the allocated cache sizes of the critical tasks are the same in each scenario. In this subsection we tackle the cache mapping problem, 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 footprints $cf_{i,q}$ of each task 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 and then we propose heuristic to solve this problem.

3.2.1 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 [6]), that is known to be NP-hard [6].

In order prove this we first recall the dynamic storage allocation problem, with the notations from [6]. 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 $\sigma : A \to \{1, 2, ..., D\}$ 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 $p_{q\to w}$ 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 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 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-complete.

3.2.2 Heuristic for the Cache Mapping Problem

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 $\left(\sum_{q=1}^{Z} \sum_{T_i \in \Psi} c_{i,q} = \Gamma\right)$, 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 $\{\Psi_m\}_{(m=1,2,\ldots,U)}$, $\bigcup_{m=1}^{U} \Psi_m = \mathcal{T}$, $\Psi_m \cap \Psi_k = \{\varnothing\}, m \neq k$. In order to build the $\{\Psi_m\}$ subsets we have to generate all possible tasks subsets and we 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 $C_N^1 + C_N^2 + \ldots C_N^{\frac{[N+1]}{2}}$, 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. Fig. 6 presents the cache of the 4 tasks in the 3 scenarios, for a possible cache map. As also visible in Fig. 6, 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 parts are always the same). As a result we now have two similar instances

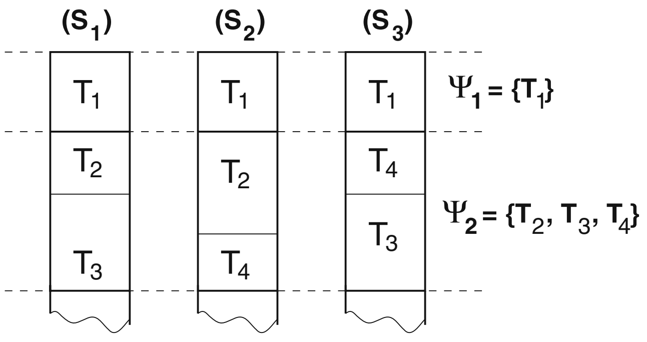


Figure 6 Example: L2 cache footprints.

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.

We define the CCR_i of a task T_i as being its cache content reuse:

$$CCR_i = \sum_{\substack{S_q \rightarrow S_w \\ c_{i,q}=c_{i,w} \\ b_{i,q}=b_{i,w}}} c_{i,q} \cdot p_{q\rightarrow w} + \sum_{\substack{S_q \rightarrow S_w \\ c_{i,q}>c_{i,w} \\ b_{i,q}=\not\approx b_{i,w}}} c_{i,w} \cdot p_{q\rightarrow w} \qquad (1)$$

The first sum corresponds to the case when $c_{i,q} = c_{i,w}$ and the second sum corresponds to the case when $c_{i,q} > c_{i,w}$, under the conditions introduced in Section 3.1.

For the general case in which Ψ_m contains N_m tasks, our mapping heuristic is described by Algorithm 1. 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,

Algorithm 1 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 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 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 Fig. 6). 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 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.

If Algorithm 1 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 (or they are performed simultaneously). This is an interesting problem by itself, and it is a 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 $p_{q\rightarrow w}$ 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 $p_{q\rightarrow w} = \frac{1}{Z}$ (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 1 the complexity of finding 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.

3.3 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 3.1, 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 it is not 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. Moreover the lines caching code should be distinguished from the lines that cache data (in general L2s are unified). 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).

4 Experimental Results

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

The considered experimental platform is CAKE, having four identical Trimedia processor cores running in parallel, each core having its own L1 data and instructions cache, and all the cores sharing the L2. In Table 1 we present for the Trimedia L1 and for the L2 the size, associativity, line size, access latency, and when applicable, the number of blocks. The latency accessing the L2 includes the interconnection network overhead, and the latency to the off-chip memory is 110 cycles.

On the CAKE platform we run a workload consisting of six applications composed of various media tasks, most of them deriving from the MediaBench suite [4]. From the MediaBench we pruned out the programs that are relatively small and not memory intensive. Moreover, to make the benchmarks more representative for emerging technologies, we added two H.264 video processing programs, an encoder and a decoder. As a result we exercised the following programs: H.264, MPEG2, EPIC, audio, and JPEG, all encoders and decoders. An application is formed by a collection of four such programs, each of which representing a task. In Table 2 we present the L2 cache working set for each task, for both the instructions and the data of the task. To determine these working sets, we perform the following experiment. We simulate each task in isolation with several L2 cache sizes, partitioning the L2 for the task's instructions and data. In different sim-

Table 1 CAKE cache configuration.

L1 cache	Data: 16 KB, 8 ways, 64 bytes line size, 3 cycles access latency
	Instructions: 32 KB, 8 ways, 64 bytes line size, 3 cycles access latency
L2 cache	Unified: 512 KB, 4 ways, 512 bytes line size, 12 cycles latency, 4 banks

ulations, we allocate for the task's instructions and data different power of two numbers of L2 sets, as required by the static partitioning methods described in Section 2. For each task's instructions and data, the values in Table 2 represent the number of L2 sets after which the misses experimented saturate, meaning that an doubling in cache size brings less than 5% decrease in number of L2 misses. As one can see in Table 2 the tasks working sets are relatively large, when compared with the total considered L2 size (the entire cache having 256 sets), for 4 of the tasks even reaching the entire cache size. Thus these tasks are relevant for the present L2 cache investigation.

Using different combinations of these tasks, we build 6 different applications ($A_1 \ldots A_6$), as visible in Table 3. Each application has 7 execution scenarios (chosen at random from the total set of possible tasks combinations) and one or two critical tasks. The applications are executed one by one, doing a simulation per application, per scenario switching rate. For each scenario switching rate, the application passes through a random sequence of scenarios. For each application, the tasks that execute in parallel and the sizes of their cache partitions are illustrated in Table 3, for each scenario. The cache sizes are measured in number of sets. As for each scenario we use the static partitioning method introduced in [16] the cache of a task can be partitioned among its data and its instructions, hence Table 4 presents these values. The first value corresponds to the task's data cache partition and the second to the instructions, or it contains an "u" if the cache partition is unified for data and instructions. The critical tasks of each application are illustrated in bold.

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

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, ranging from 100 Hz (one switch every 0.01 second) to 1 Hz (one switch every second). In Fig. 7 we present the average et^{cr} variations over all the critical tasks of each exercised application. For each investigated scenario switching frequency, we look at three cases: (1) the cache footprints determined with the method in Section 3.2 (*Critical task prio*), (2) the cache footprints determined with the method in Section 3.2, but giving no priority to critical task (*No critical task prio*), and (3) the conventional shared cache (*Shared*). One can observe in Fig. 7 that the variations in the average et^{cr} are very small if during the cache mapping the critical tasks have priority. These variations represent at maximum only 0.1% from the critical tasks execution time. Moreover, we would like to mention that not only the average, but also the variations of et^{cr} for each application are below 0.1%. If no priority is given to the mapping of critical tasks the et^{cr} variations increase with scenario switch frequency, reaching a relative value of 11% for a switch rate of 100 Hz. For the shared cache the relative et^{cr} variations represent 5% from the min-

Table 2 Tasks' working sets in L2 (instructions and data).

Task	Instr. size (sets)	Data size (sets)
H.264 encoder	128	128
H.264 decoder	64	256
MPEG2 encoder	1	256
MPEG2 decoder	32	256
EPIC encoder	32	128
EPIC decoder	1	128
Audio encoder	4	64
Audio decoder	64	128
JPEG encoder	2	64
JPEG decoder	1	64

Table 3 Applications and their 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 4 Applications, execution scenarios and the cache per task (data/code; u=unified).

	S_1	S_2	S_3	S_4	S_5	S_6	S_7
A_1	**JPEGe 32/u**	**JPEGe 32/u**	**JPEGe 32/u**	**JPEGe 32/u**	**JPEGd 32/u**	H264e 128/u	H264e128/64
	JPEGd 32/u	JPEGd 32/u	JPEGd 128/u	H264e 128/u	H264e 128/u	**AUDe 16/u**	–
	H264e 64/16	H264e 128/u	**AUDe 16/u**	**AUDe 16/u**	**AUDe 16/u**	–	–
	AUDe 16/u	–	–	–	–	–	–
A_2	H264d 32/32	H264d 64/32	EPICd 64/u	H264d 64/32	EPICd 32/u	H264d 128/32	H264d 64/64
	MPG2d 64/32	MPG2d 64/32	MPG2d 64/32	**EPICd 32/u**	MPG2d 128/64	**EPICd 32/u**	MPG2d 64/32
	EPICd 32/u	**EPICd 32/u**	**AUDd 32/u**	**AUDd 32/u**	–	–	–
	AUDd 32/u	–	–	–	–	–	–
A_3	**MPG2e 128/16**	**MPG2e 128/16**	**MPG2e 128/16**	**MPG2e 128/16**	**MPG2e 128/16**	**MPG2e 128/16**	**MPG2e 128/16**
	AUDe 16/u	AUDe 16/8	AUDd 32/16	JPEGe 16/u	AUDe 64/u	AUDd 64/u	–
	AUDd 32/u	AUDd 32/8	JPEGe 16/u	AUDe 32/16	–	–	–
	JPEGe 16/u	–	–	–	–	–	–
A_4	**H264e 64/u**	**H264e 64/u**	MPG2e 128/u	**H264e 64/u**	**H264e 64/u**	JPEGd 64/16	**H264e 64/u**
	MPG2e 64/32	EPICd 64/u	EPICd 64/u	EPICd 16/u	EPICd 64/u	EPICd 128/u	–
	EPICd 16/u	JPEGd 16/u	JPEGd 16/u	MPG2e 128/u	–	–	–
	JPEGd 16/u	–	–	–	–	–	–
A_5	**EPICd 64/u**	**EPICd 64/u**	**EPICd 64/u**	**EPICd 64/u**	**EPICd 64/u**	MPG2d 64/32	AUDd 64/32
	MPG2d 64/32	AUDd 32/u	MPG2d 64/32	JPEGd 32/u	AUDd 128/u	AUDd 64/32	JPEGd 64/32
	AUDd 16/u	MPG2d 64/32	JPEGd 32/u	AUDd 64/32	–	–	–
	JPEGd 16/u	–	–	–	–	–	–
A_6	**H264d 64/u**	**H264d 64/u**	**H264d 64/u**	**H264d 64/u**	MPG2e 128/u	**H264d 64/u**	**JPEGe 16/u**
	AUDe 32/8	AUDe 64/32	AUDe 32/16	AUDe 64/32	AUDe 64/u	MPG2e 128/u	–
	JPEGe 16/u	**JPEGe 16/u**	MPG2e 64/8	**JPEGe16/u**	–	–	–
	MPG2e 64/8	–	–	–	–	–	–

imum *Shared* et^{cr}, and we notice no clear dependence among the switching rate and et^{cr}. Furthermore, the et^{cr} is, on average, with 13% larger in the shared cache case, than in the repartitioned cache one. This indicates that the repartitioning is improving the system performance.

Apart from critical tasks execution time variations, the other metric used for evaluating the compositionality is the number of inter-task conflicts. For each task T_i we define the number of inter-task conflicts as the number of times another task flushes some T_i data out of the cache. In a repartitioned cache, these conflicts occur as a results of the late cache flush policy presented in the previous section. The number of inter-task conflicts of an application is the sum of the conflicts suffered by each of the application tasks. Table 5 illustrates the number of conflict misses for each of the 6 applications, per scenario switching frequency. The values presented in Table 5 are relative to the corresponding application total number of misses. The last row represents an average value of all the applications inter-task conflicts.

We investigate scenario switching frequencies from 100 Hz to 1 Hz. These conflict misses are presented for two cases: conventional shared L2 (*shared*), and set based repartitioned L2 (*repart*).

When the L2 is repartitioned and the scenarios are switched at a high frequency (20 Hz to 100 Hz), the average relative number of conflicts reaches a value of 8% (with a maximum of 11% for application A_5, for a 100 Hz scenario change frequency). For scenario switching rates under 10 Hz the percentage of inter-task conflicts is at most 4% for each application. 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 78% and the average for all applications and all frequencies is 70%. As expected, in general the percentage of inter-task conflicts decreases with a scenario switching rate decrease.

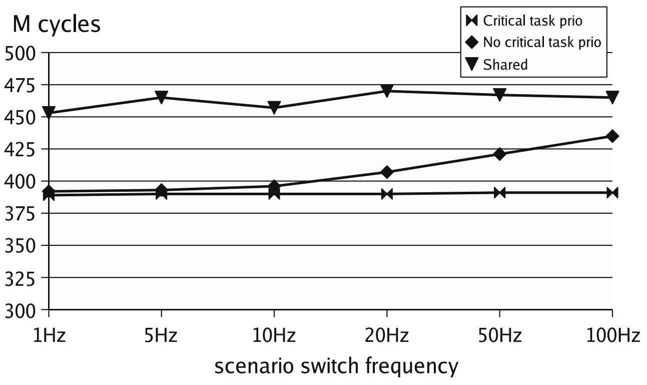


Figure 7 Average critical tasks execution time.

Table 5 Inter-task conflict misses relative to the entire application number of misses.

		100 Hz	50 Hz	20 Hz	10 Hz	5 Hz	1 Hz
A_1	Shared	70%	62%	66%	53%	54%	56%
	Repart	3%	2%	0%	0%	0%	0%
A_2	Shared	75%	74%	73%	72%	76%	74%
	Repart	14%	10%	7%	6%	4%	0%
A_3	Shared	69%	69%	71%	63%	62%	59%
	Repart	7%	4%	4%	2%	1%	0%
A_4	Shared	77%	77%	77%	78%	77%	75%
	Repart	20%	13%	12%	8%	3%	2%
A_5	Shared	76%	78%	75%	74%	72%	70%
	Repart	11%	8%	7%	4%	1%	1%
A_6	Shared	77%	74%	72%	73%	72%	75%
	Repart	6%	5%	2%	1%	0%	0%
avg	Shared	74%	72%	72%	68%	68%	68%
	Repart	10%	8%	6%	4%	2%	1%

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.

4.2 Performance

We measure the performance using two metrics: (1) the number of misses per instruction (MPI) to describe the L2 performance and (2) the processors' average cycles per instruction (CPI) to present the performance of the entire system. Because we are interested in the performance of the multimedia processing, the CPI presented in this section represent an average for the four Trimedia cores. Including the CPI of the control processor in the average would only decrease it, as the maximum MIPS CPI is 1.2, with the average of 1.05 over all the simulations experimented. Thus we can claim that the overhead of the RTCM is, as expected, negligible, because the cache mapping optimization is executed at design-time. Moreover, the data structures needed for the RTCM are small (<4 KB), as the main information the RTCM needs are the start set and partition size for each task in each scenario.

In general, two phenomena determine the difference in number of misses between a shared and a partitioned cache. If the cache is partitioned, the inter-task cache flushing is eliminated (which means less misses) but every task can use less cache space than in the shared case (which means more misses). Moreover, repartitioning the cache at run-time requires parts of the cache to be flushed, thus an extra overhead is present.

We compare the performance in the following cases: (1) a set based repartitioned L2 with the cache footprints determined with Algorithm 1 presented in the previous section (*Alg1 footprints*), (2) a set based repartitioned L2 with randomly chosen cache footprints (*Random footprints*), (3) a conventional shared L2 (*Shared*), (4) a statically set based partitioned L2 (*Static*), and (5) an infinite L2 cache. 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 8 MBytes.

Tables 6 and 7 present the MPI and the CPI values for the 6 applications per scenario switching frequency. Figures 8 and 9 graphically present the same data but averaged over all applications for the MPI and CPI, respectively. The data in the Figs. 8 and 9 represent the average CPI and MPI over the 6 applications, per scenario switching frequency. The MPI for the infinite cache is not presented as it is practically equal to 0.

When the cache mapping is performed according to the proposed method, the average over all applications and all scenario switching frequencies for the number of L2 lines flushed at each scenario transition represent 19% (with a maximum of 37% over all simulations) of the total L2 size. If the cache mapping is performed at random, this percentage increases to 36% (with a peak of 59%). Nevertheless, despite the flushing penalty, the MPI for the dynamic cache repartitioning using the proposed mapping solution is on average 44% and 60% smaller than the case when the mapping is random and the case when the L2 is shared, respectively. On average, the L2 misses reduction results in a 7% and 10% better CPI, when compared with a random L2 mapping and a shared cache, respectively. Moreover, the maximum CPI improvement that can be achieved by employing an infinite L2 is 20%. Hence the above mentioned CPI improvements brought by dynamic repartitioning represent on average 50% from the possible improvements of ideal, infite L2. We would like to underline that these improvements occur when partitioning the L2 cache, while preserving its size. Furthermore, the performance difference between the random mapped footprints and the Alg1's footprints motivates the utilization Alg1 footprint calculation.

As visible in Tables 6 and 7, when compared with a shared L2, A_3 benefits from maximum performance improvement, namely 53% MPI and 15% CPI, for a scenario switching rate of 1 Hz. In the case of A_3 the

Table 6 Applications L2 misses per instruction ($\times 10^2$) function of the scenario change frequency.

		1 Hz	5 Hz	10 Hz	20 Hz	50 Hz	100 Hz
A_1	Dynamic partitioned Alg. 1	0.008	0.008	0.009	0.096	0.099	0.011
	Dynamic partitioned Rand.	0.010	0.011	0.012	0.128	0.136	0.018
	Static partitioned	0.014	0.016	0.021	0.024	0.025	0.025
	Shared	0.010	0.093	0.013	0.013	0.012	0.016
A_2	Dynamic partitioned Alg. 1	0.048	0.048	0.049	0.060	0.081	0.110
	Dynamic partitioned Rand.	0.07	0.063	0.089	0.097	0.092	0.124
	Static partitioned	0.056	0.052	0.065	0.065	0.065	0.079
	Shared	0.212	0.103	0.101	0.102	0.107	0.110
A_3	Dynamic partitioned Alg. 1	0.260	0.268	0.280	0.300	0.340	0.452
	Dynamic partitioned Rand.	0.520	0.530	0.560	0.580	0.950	1.080
	Static partitioned	0.390	0.430	0.420	0.427	0.440	0.440
	Shared	0.560	0.590	0.710	0.880	0.960	1.010
A_4	Dynamic partitioned Alg. 1	0.720	0.730	0.830	0.820	0.870	0.870
	Dynamic partitioned Rand.	0.810	0.800	0.842	0.850	0.950	1.220
	Static partitioned	0.850	0.860	0.850	0.870	0.880	1.090
	Shared	1.120	1.130	1.160	1.200	1.190	1.350
A_5	Dynamic partitioned Alg. 1	0.310	0.455	0.495	0.532	0.628	0.836
	Dynamic partitioned Rand.	0.652	0.587	0.689	0.672	0.850	1.290
	Static partitioned	0.630	0.652	0.800	0.796	0.916	1.349
	Shared	1.440	1.431	1.429	1.471	1.492	1.670
A_6	Dynamic partitioned Alg. 1	0.031	0.035	0.041	0.056	0.069	0.078
	Dynamic partitioned Rand.	0.058	0.063	0.064	0.072	0.087	0.093
	Static partitioned	0.062	0.068	0.074	0.074	0.075	0.076
	Shared	0.073	0.120	1.290	1.360	1.580	1.790
avg	Dynamic partitioned Alg. 1	0.229	0.257	0.284	0.311	0.348	0.393
	Dynamic partitioned Rand.	0.458	0.437	0.509	0.545	0.649	0.638
	Static partitioned	0.427	0.346	0.372	0.376	0.400	0.510
	Shared	0.552	0.578	0.784	0.841	0.894	0.996

critical task owns half of the cache, thus this half benefits from full cache content reuse at scenario switch. In contrast A_4 experiences the smallest performance improvement among the 6 applications, 15% MPI and 8% CPI, for the same scenario switching rate. Nevertheless the maximum possible performance improvement for A_4 is 16% leaving not so much room for improvement, whereas for A_3 this is 21%. One can notice that dynamic repartitioning delivers for both applications more than half of the possible performance boost.

When comparing with a statically partitioned cache the proposed method exhibits, on average, 25% less misses per instruction, leading to 4% better CPI. The performance differences in MPI and CPI among the static and dynamic partitioned cache decrease with the increase of scenario switching frequency. For a scenario switching rate of 100 Hz the dynamically partitioned L2 outperforms the statically partitioned one with 1% for the CPI metric and 23% for the metric MPI, whereas for a scenario switching rate of 1 Hz the improvement is 7% and 47%, respectively. These 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.

Again, some applications are more sensitive to cache repartitioning than others. For example applications A_3 and A_6 experience up to 8% and 5% better CPI, with a corresponding 33% and 50% MPI reduction, when choosing for dynamic repartitioning instead of the simple static partitioning. The smallest difference among the static and dynamic partitioning is present for A_2. There dynamic partitioning brings only 1% better CPI and 14% MPI reduction. In general we observed no direct correlation among the static/dynamic performance difference and the cache footprints. The main difference among the static and the dynamic partitioning is that, if the application executes in scenarios that have only few tasks active, the dynamic partitioning uses a larger cache fraction than the static partitioning, thus it performs better. In general we observe that the performance of the statically partitioned L2 is better than the one of the shared cache, and, most important,

Table 7 Applications cycles per instruction function of the scenario change frequency.

		1 Hz	5 Hz	10 Hz	20 Hz	50 Hz	100 Hz
A_1	Dynamic partitioned Alg. 1	1.25	1.27	1.28	1.19	1.31	1.32
	Dynamic partitioned Rand.	1.36	1.37	1.36	1.32	1.47	1.50
	Static partitioned	1.28	1.29	1.32	1.32	1.33	1.36
	Shared	1.40	1.39	1.40	1.41	1.40	1.42
	Infinite size	1.24	1.24	1.24	1.24	1.24	1.24
A_2	Dynamic partitioned Alg. 1	1.26	1.27	1.28	1.28	1.31	1.31
	Dynamic partitioned Rand.	1.35	1.34	1.36	1.39	1.38	1.43
	Static partitioned	1.27	1.27	1.28	1.29	1.29	1.30
	Shared	1.40	1.48	1.42	1.44	1.44	1.48
	Infinite size	1.12	1.12	1.12	1.12	1.12	1.12
A_3	Dynamic partitioned Alg. 1	1.16	1.16	1.18	1.23	1.26	1.31
	Dynamic partitioned Rand.	1.34	1.35	1.37	1.38	1.41	1.42
	Static partitioned	1.27	1.29	1.28	1.28	1.29	1.29
	Shared	1.37	1.37	1.40	1.42	1.42	1.42
	Infinite size	1.07	1.07	1.07	1.07	1.07	1.07
A_4	Dynamic partitioned Alg. 1	1.18	1.19	1.21	1.21	1.24	1.25
	Dynamic partitioned Rand.	1.21	1.21	1.23	1.23	1.26	1.28
	Static partitioned	1.24	1.25	1.24	1.24	1.25	1.2
	Shared	1.28	1.27	1.28	1.28	1.27	1.29
	Infinite size	1.07	1.07	1.07	1.07	1.07	1.07
A_5	Dynamic partitioned Alg. 1	1.32	1.32	1.32	1.33	1.37	1.40
	Dynamic partitioned Rand.	1.33	1.35	1.38	1.38	1.41	1.43
	Static partitioned	1.38	1.37	1.39	1.39	1.40	1.43
	Shared	1.46	1.48	1.47	1.48	1.48	1.52
	Infinite size	1.12	1.12	1.12	1.12	1.12	1.1
A_6	Dynamic partitioned Alg. 1	1.17	1.18	1.20	1.23	1.25	1.27
	Dynamic partitioned Rand.	1.23	1.24	1.24	1.26	1.29	1.29
	Static partitioned	1.24	1.25	1.27	1.27	1.27	1.27
	Shared	1.28	1.31	1.32	1.32	1.41	1.43
	Infinite size	1.10	1.10	1.10	1.10	1.10	1.10
avg	Dynamic partitioned Alg. 1	1.21	1.23	1.24	1.25	1.29	1.31
	Dynamic partitioned Rand.	1.30	1.31	1.32	1.33	1.37	1.39
	Static partitioned	1.29	1.29	1.30	1.30	1.30	1.32
	Shared	1.36	1.38	1.38	1.39	1.40	1.43
	Infinite size	1.12	1.12	1.12	1.12	1.12	1.12

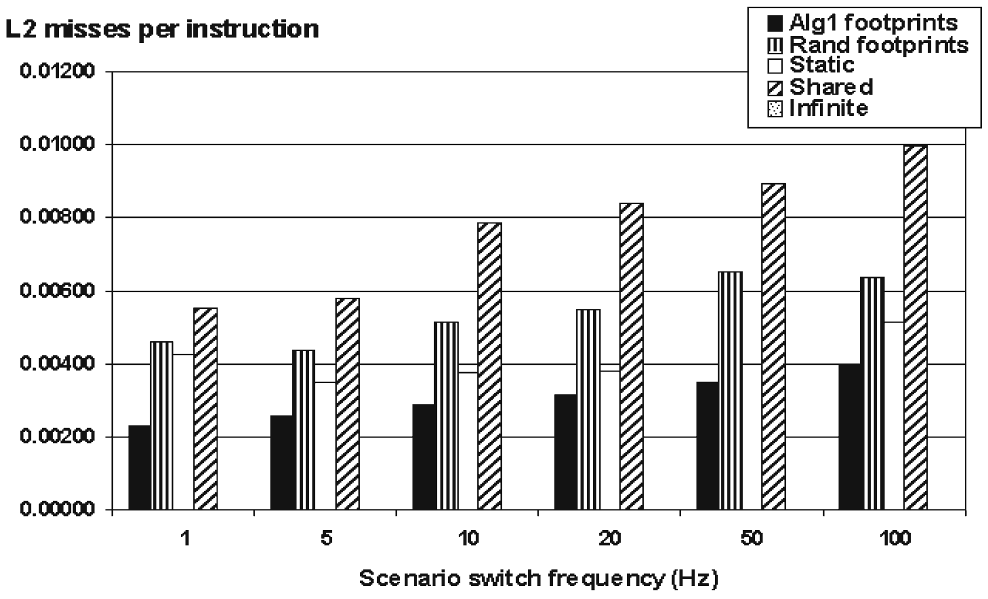
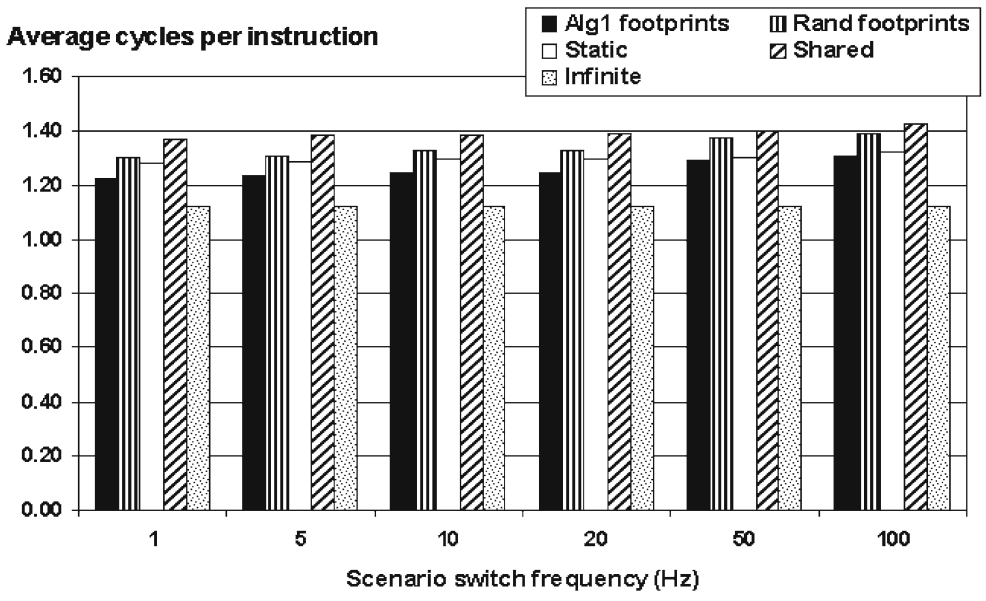


Figure 8 Performance: L2 misses per instruction.

Figure 9 Performance: cycles per instruction.



the performance of the dynamically partitioned cache is the best, better than both the static partitioned and the shared one.

When looking solely at the dynamic partitioning involving Algorithm 1, we can notice that for the Figs. 8 and 9 that the MPI increases with 40% when the scenario switching frequency varies from 1 Hz to 100 Hz. As a result the CPI increases with 6% for the same scenario switching range. However, for realistic scenario switching ranges (under 10 Hz) the difference in MPI is on average 18% and in CPI is 1%. This suggests that in such a case the cache flushing penalty is negligible.

5 Related Work

In this section we discuss two main related research topics: dynamic cache partitioning in general and caches for embedded domain in particular.

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 cache management receives more and more attention. In the general field of multiprocessors several authors tackle dynamic cache partitioning [5, 10, 22]. In [5] 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 previously conventional schemes. In [22] the authors explore existing adaptable caching strategies that balance cache demand of each task and improve the overall system throughput. In [10] the authors introduce a cache partitioning strategy to improve fairness (defined as how uniform tasks are slowed down due to cache sharing). All these proposals bring interesting ideas to the field. However compositionality and critical tasks performance protection are not targeted, as these papers are positioned in the general purpose computation area.

Different approaches that attempt to make caches usable in real-time environments exist and were described in the literature. These approaches fall into two categories: (1) attempts to estimate the tasks cache behavior, and (2) attempts to partition the caches among tasks such that they become “more” predictable.

To the best of our knowledge the only articles that investigate the impact of cache sharing in a multiprocessor architecture are [1] and [2].

In [1] 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 a parallel execution than in a sequential execution of the same computation. This paper provides an analytical framework to calculate the extra cache size needed in a parallel execution in order to obtain the same off-chip data traffic as in a sequential execution. This method is based on a restricted computation model which assumes that a computation is represented as a Directed Acyclic Graph (DAG) of tasks synchronized with barriers and scheduled in a depth-first manner. Nevertheless, real life media applications are difficult to express or implement using DAGs scheduled in such a specific way (depth-first). Furthermore the authors take into account only an ideal cache model, which makes their method difficult to apply for real applications.

The work in [2] 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 significant performance penalty to the co-scheduled tasks. This suggests that only a method to predict the extra involved penalty is not enough. Therefore the research in [2] makes the point of this paper even stronger, as it highlights the need for methods to manage caches in a multiprocessor environment.

The second manner to deal with cache unpredictability, cache partitioning, was investigated by several research groups. Cache partitioning has been proposed to: (1) improve performance [19, 23] and (2) acquire predictability [11, 12, 24].

The authors of [23] published several papers thoroughly presenting a method in which a task dynamically "steals" cache ways from other tasks, for the purpose of improving the performance. The authors of [24] propose to allocate more cache ways to high priority tasks. However, as already mentioned, in the context of compositionality, the main shortcoming of associativity based approaches is that the number of allocable resources is restricted to the number of ways in a set (typically small) thus only a limited number of tasks can be accommodated.

The authors of [8] and [19] propose a compositional data (instruction) cache organization. The cache is analyzed and a partitioning is decided at compile time and imposed at run-time by specific cache instructions. The result is that this scheme outperforms a conventional cache, while ensuring compositionality. The main drawback of this approach is that the underlying analysis is difficult in the multiprocessor case, as the detailed task timing and synchronization has to be known at design time, which is usually not the case.

In [18] the cache is partitioned among tasks at compile and link time. In [11] the authors propose to divide the cache among each real-time task. For non-real-time tasks a shared cache pool is provided. The authors of [12] propose an operating system controlled cache partitioning. However, none of these proposals can be utilized in our case, as they provide a static solution, while we are considering dynamic applications which have multiple execution scenarios depending on the user requests.

In [9] the authors present a quality of service like cache management strategy for multiple memory access streams, based on a priority scheme. The paper reports interesting performance enhancements, however compositionality is not the target of their work.

In conclusion, to the best of our knowledge, the existing cache management methods cannot cope efficiently with applications that demand compositionality, have critical tasks, and dynamically switch among different utilization scenarios.

6 Conclusions

In this paper we proposed a dynamic cache management method that enhances compositionality for multimedia applications with multiple utilization scenarios. This method aims at multiprocessor platforms that comprise shared caches. The dynamic repartitioning requires 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 proposed 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 (100 Hz to 1 Hz). The workload consisted of six applications formed by various task from the MediaBench suite augmented with an H.264 algorithm. We applied the repartitioning method to the CAKE's L2 cache that is shared among the processors, therefore highly exposed to inter-task interferences. 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 it reaches 68%. Moreover, the relative variations of critical tasks execution time are less than 0.1%, over the entire scenario switching frequency range studied. With respect to performance, the dynamic repartitioning reduces the off-chip memory traffic on average with 60%, when compared with the shared cache. As a consequence, the average number of cycles needed to execute an instruction is decreased with 10%, when compared with the shared cache, under the circumstances that a maximum of 20% reduction is potentially achievable when using an infinite L2 cache. In general we observe that the performance of the statically partitioned L2 is better than the one of the shared cache, and, most important, the performance of the dynamically partitioned cache is the best, better than both the static partitioned and the shared one. Therefore, despite the involved cache flushing, the repartitioned L2 enables high compositionality and performs better than the shared cache.

References

1. Blelloch, G. E., & Gibbons, P. B. (2004). Effectively sharing a cache among threads. In *Proceeding of SPAA* (pp. 235–244).
2. Chandra, D., Guo, F., Kim, S., & Solihin, Y. (2005). Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceeding of HPCA* (pp. 340–351).
3. Chiou, D. T. (1999). *Extending the reach of microprocessors: Column and curious caching*. PhD thesis, Department of EECS, MIT, Cambridge, MA.
4. Chunho, L., Potkonjak, M., & Mangione-Smith, W. (1997). Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings, international symposium on microarchitecture*.
5. Dybdahl, H., & Stenstrom, P. (2007). An adaptive shared/private nuca cache partitioning scheme for chip multiprocessors. In *Proceeding of IEEE international symposium of high performance computer architecture* (pp. 2–12).
6. Garey, M. R., & Johnson, D. S. (1979). *Computers and intractability: A guide to the theory of NP-completeness*. New York: W. H. Freeman.
7. Hennesy, J. L., & Patterson, D. A. (2003). *Computer architecture: A quantitative approach*. San Fransisco: Morgan Kaufmann.
8. Irwin, J., May, D., Muller, H., & Page, D. (2002). Predictable instruction caching for media processors. In *13th International conference on application-specific systems, architectures and processors (ASAP)* (pp. 141–150).
9. Iyer, R. (2004). Cqos: A framework for enabling qos in shared caches of cmp platforms. In *Proceeding of the 18th annual international conference on supercomputing* (pp. 257–266).
10. Kim, S., Chandra, D., & Solihin, Y. (2004). Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceeding of IEEE PACT* (pp. 111–122).
11. Kirk, D. B. (1989). Smart (strategic memory allocation for real-time) cache design. In *IEEE symposium on real time systems* (pp. 229–237).
12. Liedtke, J., Härtig, H., & Hohmuth, M. (1997). Os-controlled cache predictability for real-time systems. In *3rd IEEE real-time technology and applications symposium*.
13. Molnos, A. (2008). *Task centric memory management for an on-chip multiprocessor*. PhD Thesis, Technical University of Delft (to appear).
14. Molnos, A., Heijligers, M., Cotofana, S., & van Eijndhoven, J. (2004). Compositional memory systems for data intensive applications. In *Proceedings, design, automation and test in Europe* (pp. 728–729).
15. Molnos, A., Heijligers, M., Cotofana, S., & van Eijndhoven, J. (2005). Compositional memory systems for multimedia communicating tasks. In *Proceedings, DATE*.
16. Molnos, A., Heijligers, M., Cotofana, S., & van Eijndhoven, J. (2006). Compositional, efficient caches for a chip multiprocessor. In *Proceedings, design, automation and test in Europe*.
17. Moore, G. (1965). Cramming more components on integrated circuits. *Electronics*, April 19.
18. Mueller, F. (1995). Compiler support for software-based cache partitioning. *ACM SIGPLAN Notices*, *30*(11), 137–145.
19. Muller, H., Page, D., Irwin, J., & May, D. (2002). Caches with compositional performance. In *Proceedings, embedded processor design challenges* (pp. 242–259).
20. Nayfeh, B. A., & Olukotun, K. (1994). Exploring the design space for a shared-cache multiprocessor. In *21st Annual Int. Symp. Computer Architecture* (pp. 166–175).
21. Sebek, F. (2001). *The state of the art in cache memories and real-time systems*. MRTC Technical Report (01/37).
22. Settle, A., Connors, D., Gibert, E., & González, A. (2006). A dynamically reconfigurable cache for multithreaded processors. *Journal of Embedded Computing, 2*, 221–233.
23. Suh, G. E., Rudolph, L., & Devadas, S. (2004). Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, *28*(1), 7–26.
24. Tan, Y., & Mooney, V. (2003). A prioritized cache for multitasking real-time systems. In *Proceedings of the 11th workshop on synthesis and system integration of mixed information technologies* (pp. 168–175).
25. Terechko, A. (2005). *Hardware cache coherence prototyping for the tm2270 trimedia*. Philips Research Technical Note PR-TN 2005/00312.
26. van Eijndhoven, J. T., Hoogerbrugge, J., Jayram, M., Stravers, P., & Terechko, A. (2005a). Cache-coherent heterogeneous multiprocessing as basis for streaming applications. In *Dynamic and robust streaming between connected ce-devices*. Boston: Kluwer.
27. van Eijndhoven, J. T., Hoogerbrugge, J., Jayram, M., Stravers, P., & Terechko, A. (2005b). *Dynamic and robust streaming between connected CE-devices.* Boston: Kluwer.

Anca M. Molnos received the M.S. degree in computer science from the "Politehnica" University of Bucharest, Romania in 2001. Between 2002 and 2006 she was a Ph.D student at the Delft University of Technology, Delft, The Netherlands, working on cache management for embedded multi-processor systems executing multimedia applications. Since 2006, she is a senior scientist at NXP Semiconductors, The Netherlands. Her research interests include performance analysis for real-time distributed systems, resource management for embedded multi-processors and low power scheduling techniques.

Marc J.M. Heijligers received his Masters in Information Technology from the Eindhoven University of Technology in 1991, and obtained his PhD in 1996. In his PhD thesis he investigated the use of optimization techniques for architectural synthesis. He joined Philips Research in 1996, working on architectural synthesis for high-throughput video processing, then on he worked on C++ code analysis for IC design in the Electronic Design & Tools department. Since 2001, he is leader of the Systems on Silicon Integration cluster, working on topics such as low power design, cache management for multi-processor systems, embedded FPGAs, channel decoding, smart image processing, and strategical documents in the area of the semiconductors industry. Since September 2006, Marc is working at NXP Semiconductors, The Netherlands, where he is group manager of the IC-lab digital VLSI group.

Sorin D. Cotofana received the M.S. degree in computer science from the "Politehnica" University of Bucharest, Romania, and the Ph.D. degree in electrical engineering from the Delft University of Technology, Delft, The Netherlands. He worked for a decade with the Research and Development Institute for Electronic Components (ICCE), Bucharest. He is currently an Associate Professor with the Computer Engineering Laboratory at Delft University of Technology. His research interests include various topics from nano-electronics and nano-device specific design methodologies and computational paradigms, fault tolerant computing, embedded systems, reconfigurable computing, computer arithmetic, low power hardware, and multimedia and vector architectures and processors. Dr. Cotofana is a member of the IEEE Computer Society and of IEEE Circuits and Systems Society.

Jos T.J. van Eijndhoven is co-founder of Vector Fabrics BV, developing tools for HW/SW co-design. Before 2007, he was principal architect at NXP Semiconductors Research, working on programmable multimedia hardware architectures and the associated mapping of media processing applications. From 1984 to 1998 he was senior research member in the Design Automation group at the Eindhoven University of Technology, The Netherlands. In 1986 he spent a sabbatical at the IBM Thomas J. Watson Research Laboratory, Yorktown Heights, New York, USA pioneering the research on high level synthesis. He studied Electrical Engineering at the Eindhoven University of Technology, The Netherlands, obtaining his Ph.D. degree in 1984 for research on mixed-level simulation. Jos van Eijndhoven coauthored about 100 scientific publications and currently holds 15 worldwide patents.