Compositional, dynamic cache management for embedded chip multiprocessors

Anca M. Molnos, Marc J.M. Heijligers
NXP Semiconductors / Corporate I&T
HTC 37, Eindhoven, The Netherlands
{anca.molnos, marc.heijligers}@nxp.com

Sorin D. Cotofana
Technical University of Delft
Mekelweg 4, Delft, The Netherlands
s.d.cotofana@ewi.tudelft.nl

Abstract

This paper proposes a dynamic cache repartitioning technique that enhances compositionality on platforms executing media applications with multiple utilization scenarios. The repartitioning among scenarios requires a cache flush, thus two undesired effects may occur: (1) the execution of critical tasks may be disturbed and (2) 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 task, such that it creates the premises for critical tasks safety, and reduces the amount of required flush, and (2) enforces these footprints and further decreases the flush penalty, at run-time. 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. For the repartitioned cache we found on average that: (1) the relative variations of critical tasks execution time are less than 0.1%, regardless the scenario switching frequency, (2) for realistic scenario switching frequencies the inter-task cache interference is at most 4% , and (3) the off-chip memory traffic reduces with 60%, and the performance (in cycles per instructions) enhances with 10%, when compared with the shared cache.

1 Introduction

Over the last years, the size and complexity of multimedia applications have a clear tendency to increase, demanding more and more performance from the underlying hardware. In the embedded field a common practice to boost performance is to use several processors integrated on a single chip, (Chip Multi-Processors - CMP). Nevertheless the speed gap between the processors and the off-chip memory widens with 50% every year. Therefore, to mitigate this gap, a CMP typically comprises a number of memory buffers.

Cache hierarchies 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 [10]. Furthermore, we assume that such a CMP executes a software application consisting of a given tasks set. 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's data at random. Unpredictability constitutes a major problem for media applications for which the completion of tasks before their deadlines is of crucial importance. Ideally, to be able to predict the overall application performance, 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 to diminish the inter-tasks interference [5, 13]. Existing work targets applications composed of tasks that all execute for the entire application's lifetime. However, a typical multimedia application may have multiple utilization scenarios (not all tasks are continuously active). For example, 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 behavior, while preserving a certain degree of compositionality.

In this paper we propose a strategy to dynamically repartition the cache at scenario changes, 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 at a scenario switch. In order to keep data correctness, the cache repartitioning implies flushing, therefore a time penalty. This is especially critical for tasks which 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) the number of necessary flushes is 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 on a CAKE platform [14] with 4 Trimedia cores executing 6 multimedia parallel applications, and we investigate a wide scenario switching frequency range (from 100Hz to 1Hz). We found that for realistic switching frequencies under 10Hz the inter-task cache interference is at most 4% for the repartitioned cache, indicating that the proposed strategy achieves high compositionality. Overall, the observed cache interference is at most 11%. Moreover, the relative variations of critical tasks execution times are less than 0.1%, for all the studied case, thus

978-3-9810801-3-1/DATE08 © 2008 EDAA

the critical tasks remain undisrupted. In addition, on average, the dynamic repartitioning reduces the off-chip memory traffic 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 per instruction is decreased with 10%, and 4%, respectively.

This paper is organized as follows. Section 2 introduces the considered multiprocessor architecture, the possible cache partitioning types, and discusses existing work. The proposed cache repartitioning method is presented in Section 3. The experimental results are presented in Section 4, and Section 5 concludes the paper.

2 Background and related work

The envisaged multi-processor architecture (Figure 1), comprises several media processors and a control processor. These processors are connected to on-chip memory banks by a fast, high-bandwidth interconnect. The memory hierarchy is organized as follows: first there are the L1 caches private to each processor, then 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. All the L1s and the L2 are maintained coherent.

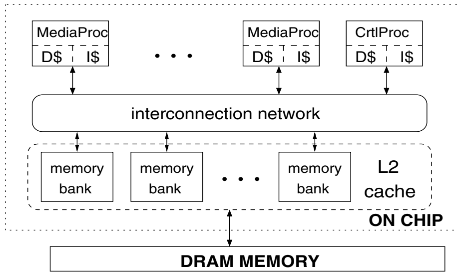

Figure 1. Multi-processor target architecture

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 $\mathcal{T}_q \subseteq \mathcal{T}$ are active. Though in this paper we consider soft real time applications, some task may be less tolerant to disturbance than others. For example in a device able to simultaneously record a video stream and play another stream, 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 tasks that cannot tolerate disturbances as critical.

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 way 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.

An existing manner to induce compositionality is to assign to each task an exclusive cache part. A conventional set associative cache is logically organized as a matrix of sets (rows) and ways (columns). To determine if a datum is cached, a set is directly addressed by a part of the datum's address, and all the ways of that set are searched. With respect to conventional cache organization we identify two possible types of partitioning: (1) associativity based, also called column caching [1] (Figure 2a): a task gets a number of cache ways from every set; and (2) set based (Figure 2b): a task gets a number of cache sets.

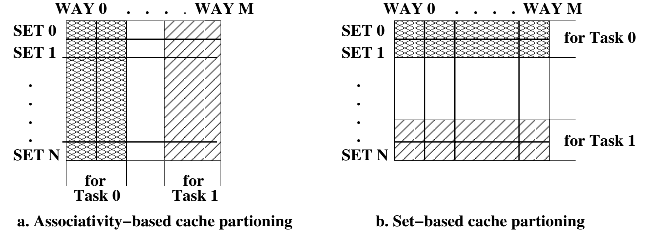

Figure 2. Types of cache partitioning

The associativity based partitioning is mostly used in the literature [12], [13] 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 a media applications there might be more tasks. If there are not enough ways, multiple tasks would share the same way, leading to unforeseeable cache interference, hence to an un-compositional system. Moreover, it is known that the performance of a program degrades when the associativity of its cache decreases.

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. In the following we discuss approaches using this type of partitioning. The authors of [9] propose a compositional cache organization. The cache is analyzed and a partitioning is decided at compile time and imposed at run-time by specific cache instructions. This scheme outperforms a conventional cache, while ensuring compositionality. However the underlying analysis is difficult in the multiprocessor case, as the detailed tasks' timing and synchronization has to be known at design time, which is usually not the case. In [8] the cache is partitioned among tasks at compile and link time. In [5] the authors propose to divide the cache among each real-time task. The authors of [6] propose an operating system controlled cache partitioning. In [7] a scheme for static set based partitioning is described. 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 the general field of multiprocessors several authors tackle dynamic cache partitioning [3], [11]. In [3] 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. In [11] the authors explore existing adaptable caching strategies that balance cache demand of each task. These proposals bring interesting ideas, however compositionality and critical tasks performance protection

are not targeted, as the purpose of these schemes is to increase the overall multiprocessor's throughput.

Due to the fact that typically in a cache there are thousands of sets and only few ways, the set based partitioning can potentially induce compositionality, therefore this is the partitioning type we consider and discuss further.

3 Dynamic cache repartitioning

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 cache are numbered from 1 to C. We define the cache footprint of T_i in the scenario S_q as the contiguous cache interval allocated to T_i, $cf_{i,q} = [b_{i,q}, b_{i,q} + c_{i,q})$, where $b_{i,q} \in [1, C - c_{i,q}]$ represents the cache unit where T_i 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 isolates the tasks in the cache to enhance compositionality. Orthogonal with 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: (1) the *cache allocation problem*, $\mathcal{CAP}$ (find the sizes c_i), and (2) the *cache mapping problem*, $\mathcal{CMP}$ (find the footprints cf_i).

Static partitioning methods consider the cache space as being uniform, in the sense that the application's 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 switch $S_q \rightarrow S_w$, the repartitioning costs may depend on $cf_{i,q}$ and $cf_{i,w}$. In the case of set based partitioning, T_i data have to be relocated into the new T_i's cache part at the scenario switch via flushing or other strategy that typically involves an overhead. In conclusion, in dynamic repartitioning the system performance heavily relies on $cf_{i,q}$ and $cf_{i,w}$, therefore the *cache mapping* becomes important.

This paper presents a dynamic cache management strategy consisting of two parts. First 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 tasks and a minimal cache repartitioning penalty. Second, we introduce a run time strategy able to impose the statically determined footprints and to further decrease the amount of flushing.

3.1 Cache mapping problem

In this subsection we first investigate the cases when the cache content can be reused, at scenario change, and then we propose an heuristic to determine the cache footprints. 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 the most recent data value is. Such mechanisms are in principle possible but in order to minimize the hardware overhead we do not embed them in our current proposal. For now we use cache flushing into the off-chip memory, to ensure data correctness. 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. To illustrate the cache reuse at a scenario change $S_q \rightarrow S_w$, we consider a task T_i active in both scenarios. For simplicity, we assume that in both footprints $cf_{i,q}$ and $cf_{i,w}$ begin on the same set ($b_{i,q} = b_{i,w}$) and the cache sizes vary with a factor of 2. Then there are two possibilities, as follows:

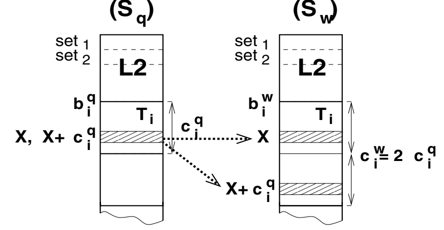

Figure 3. Doubling a task cache

(1) The cache doubles ($c_{i,w} = 2 \times c_{i,q}$), as in Figure 3. In S_q the data at address X maps in cache in $set_X = b_{i,q} + X\%c_{i,q}$, the same as the ones at address $X + c_{i,q}$. In S_w the data at address X still maps in set_X, but the data at $X + c_{i,q}$ maps in $set_X + c_{i,q}$. Thus, not all data in the $cf_{i,q}$ footprint would stay in the same location in $cf_{i,w}$. Therefore, to keep correctness, the data that does not map anymore in $cf_{i,q}$ has to be flushed. However, in order to determine which data fall into this category a search similar to the conventional cache look-up should be performed at scenario change. We do not assume the existence of such a mechanism, thus for the present work the entire $cf_{i,q}$ is flushed.

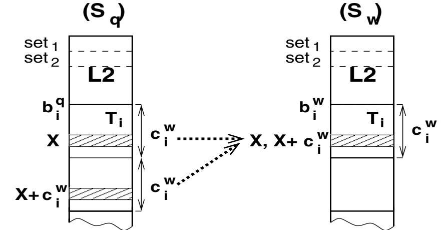

Figure 4. Halving a task cache

(2) The cache halves ($c_{i,q} = 2 \times c_{i,w}$). As visible in Figure 4, 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 S_w (for those data items $X\%c_{i,q} = X\%c_{i,w}$). However, the data in S_q for which $X\%c_{i,q} > c_{i,w}$ (for instance $X + c_{i,w}$, as illustrated in Figure 4) are relocated in $cf_{i,w}$. Thus, in order to keep data correctness, the second half of $cf_{i,q}$ has to be flushed.

A similar rationale applies when a tasks cache size increases or decreases with a factor of 2^K in two consecutive scenarios. In conclusion, on an $S_q \rightarrow S_w$ switch, there are two cases when the cache content of a task T_i can be reused: (1) if T_i cache footprint stays the same, and (2) if T_i number of cache sets decreases, and if the starting set of the new cache footprint $b_{i,w} = b_{i,q} + \varkappa \cdot c_{i,w}$, $\varkappa \in \mathbb{N}$, $\varkappa < c_{i,q}/c_{i,w}$. We denote the footprint set of a critical task as "sane" if its content is fully reused at each possible scenario change, thus $\forall q, \forall w, cf_{i,q} = cf_{i,w}, T_i \in \mathcal{T}_q, T_i \in \mathcal{T}_w$ (i.e. a critical task's data should be cached always in the same place).

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 in [7] 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. We formulate $\mathcal{CMP}$ as follows: given: (1) an application A 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 $c_{i,q}$, 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. This problem is similar with the Dynamic Storage Allocation Problem $\mathcal{DSAP}$ [4], which is an NP-complete problem. Intuitively, the caches $c_{i,q}$ correspond to the size of the item to be stored in $\mathcal{DSAP}$ and the scenarios sequence when a task is active corresponds to the arrival time and departure time of an item. As the $\mathcal{DSAP}$ is an NP-complete problem, we can infer that $\mathcal{CMP}$ is NP-complete, thus in this section we propose an heuristic to solve it.

As a first step, the $\mathcal{CMP}$ 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 $\left(\sum_{q=1}^{Z} \sum_{T_i \in \mathcal{T}_q} c_{i,q} = \Gamma\right), \forall T_i \in \Psi$, 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. The problem can be further recursively split, obtaining a set of task subsets $\{\Psi_m\}_{(m=1,2,\ldots,U)}$, $\bigcup_{m=1}^{U} \Psi_m = \mathcal{T}$, $\Psi_m \cap \Psi_k = \{\varnothing\}, \forall m \neq k$. To build $\{\Psi_m\}$ 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 $C_N^1 + C_N^2 + \ldots C_N^{\left[\frac{N+1}{2}\right]}$, 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 it is performed at design time, and typically the number of tasks is at most few tens.

We denote with CCR_i the cache content reuse of T_i:

$$CCR_i = \sum_{\substack{S_q \to S_w, c_{i,q} > c_{i,w} \\ b_{i,w} = \varkappa b_{i,q}, \varkappa \in \mathbb{N} \varkappa \leq c_{i,q}/c_{i,w}}} c_{i,w} \cdot p_{q \to w} \quad (1)$$

Considering that Ψ_m contains N_m tasks, the mapping heuristic is described by Algorithm 1. As a general line, 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 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 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 tasks have the same order in the cache in each scenario. CCR_i depends on the task position in the cache and it is recalculated at each mapping step, with the current values for $b_{i,q}$ and $b_{i,w}$. Given that a number of tasks is 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. Furthermore, $\{T_m^{cr+ok}\} \subset \Psi_m$ is the subset of critical tasks with a sane footprint if placed at the top or at the bottom of the free cache space.

Algorithm 1: Finding the cache footprint for all tasks

```
foreach Ψ_m ∈ {Ψ_m} do
    while Ψ_m ≠ {∅} do
        for T_i ∈ Ψ_m do calc. CCR_i^{t/b} and form {T_m^{cr+ok}};
        foreach {top, bottom} cache extremities do
            if {T_m^{cr+ok}} ≠ {∅} then place the
            T_i ∈ {T_m^{cr+ok}} with the largest CCR_i^{t/b};
            else place the T_i ∈ Ψ_m with the largest
            CCR_i^{t/b};
            Ψ_m = Ψ_m \ T_i;
        end
    end
end
```

If Algorithm 1 cannot sanely place all Ψ_m's critical tasks, we rerun it, but at step 5 and 6, instead of picking the task with the largest reuse we select the task with the second, third, etc. largest reuse. In the case that after all possible backtracking 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, and/or (2) the total cache size 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 subject to future research.

3.2 Run-time cache management

In order to control the cache repartitioning, we employ a software Run-Time Cache Manager (RTCM) running on the control processor. At $S_q \to S_w$, the RTCM jobs are, in order: (1) to stop the tasks that are not active in S_w and the tasks that change their footprints; 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 Subsection 3.1, (3) to update the cache partitioning tables to the new footprints, and (4) to start the S_w tasks and to resume the ones that changed their footprints.

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 occur when the flushed data are needed again in the cache. To minimize these overheads we propose a cache flushing policy is as follows:

(1) *Flush no code*. On the CAKE platform the code does not modify during execution. Thus the main memory contains a valid copy of all the application code.

(2) *Late flush*. This applies in the case a task T_i is not active in the new scenario. Only when T_i resumes its execution later in another scenario, its data are flushed out of the cache, if also T_i footprint change). In the mean time some of the data might have been already swapped out by other tasks, so some cold misses still occur, but a part of the flushing overhead is avoided. Moreover, if T_i footprint does not change, it potentially benefits from remained 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 switch when all T_i cache lines are relocated. To ensure T_i's data correctness, only the T_i's cache lines have to be flushed out of $cf_{i,q}$ (data belonging to other tasks may still be cached in some of $cf_{i,q}$ lines, from a previous execution, as allowed by the late flush).

We use the implementation of set based partitioning introduced in [7]. In addition the dynamic cache management requires each L2 line to have a task id, and the lines caching code to be distinguished from the ones caching data. Nevertheless, 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 0.2% for an L2 having 512 Bytes cache lines, when supporting 128 tasks).

4 Experimental results

In this section we investigate two issues related to cache repartitioning: the compositionality and the performance. We study them over a scenario switching rate ranging from 100Hz (one switch every 0.01 second) to 1Hz (one switch every second). The considered workload consists of 6 applications composed out of various media tasks, from the MediaBench suite [2]. from which 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 encoders and decoders: H.264, MPEG2, EPIC, audio, and JPEG, each of which representing a task. Using different combinations of these tasks, we build 6 different applications. Each application has 7 execution scenarios (chosen at random from the total set of possible task combinations) and one or two critical tasks. These applications run on a CAKE platform having 4 Trimedia processor cores and a 512KBytes L2. The access times for the different memory levels are as follows: 3 cycles for the L1, 12 cycles for the L2, and 110 cycles for the off-chip memory.

4.1 Compositionality

To evaluate compositionality, we look at the critical task execution time variations and at the number of inter-task conflicts. To check the critical task execution time (et^{cr}) variation we simulate each application with random scenario's order, and different scenario switching rates. In Figure 5 we present the average et^{cr} variations over all the critical tasks in each exercised application in three cases: (1) the cache footprints determined as in Section 3.1 (*Critical task prio*), (2) the cache footprints determined as in Section 3.1, but giving no priority to critical tasks (*No critical task prio*), and (3) the shared cache (*Shared*).

One can observe in Figure 5 that the variations of et^{cr} are very small for the case the critical tasks have mapping priority. These variations represent at maximum only 0.1% from the critical tasks execution time, thus they are practically undisturbed. If no mapping priority is given to the critical tasks, the et^{cr} variations increase with scenario switch frequency, reaching a relative value of 11% for a switch rate of 100Hz. For the shared cache the relative et^{cr} variations represent 5% from the minimum *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.

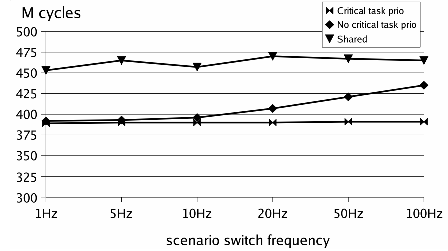

Figure 5. Avg. critical tasks execution time

We define the number of inter-task conflicts of a task T_i as the number of T_i's L2 lines flushed by another task. For the repartitioned L2 these conflicts occur as a result of our late cache flush policy. The number of inter-task conflicts of an application is the sum of the conflicts of each of A's tasks. Due to space limitation the detailed number of conflicts for each of the 6 applications are not included here. The experiments showed that 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%. When the L2 is repartitioned, for a high scenario switching frequency (20 Hz to 100 Hz), the average relative number of conflicts reach a value of 8% (with a maximum of 11%). For scenario switching rates under 10Hz the fraction of inter-task conflicts is at most 4% for each application. Thus, we can consider that a high degree of compositionality is achieved.

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. In general, two phenomena determine the number of misses' difference 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 L2 at run-time requires parts of the cache to be flushed, which might cause an extra overhead.

We compare the performance in the following cases: (1) a set based repartitioned L2 with the cache footprints determined with the Algorithm 1 in Section 3.1 (*Alg1*), (2) a set based repartitioned L2 with randomly chosen cache footprints (*Random*), (3) a shared L2 (*Shared*), (4) a statically set based partitioned L2 (*Static*), and (5) an infinite L2. 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. Due to lack of space, we present only average MPI and CPI values, in Figures 6 and 7, respectively. The MPI for the infinite cache is not presented as it equals 0.

When the cache mapping is performed according to our method, the average number of L2 lines flushed at each scenario switch represent 19% of the total L2 size. When the cache mapping is performed at random, this percentage increases to 36%. Nevertheless, despite the flushing penalty, the MPI for the L2 repartitioning using the proposed mapping is on average 44% and 60% smaller than the case when the mapping is random and the case when the L2 is shared, respectively. This results in a 7% (respectively 10%) better CPI, representing 35% (respectively 50%) from the possible improvement measured when having an infinite L2, while preserving the same cache size.

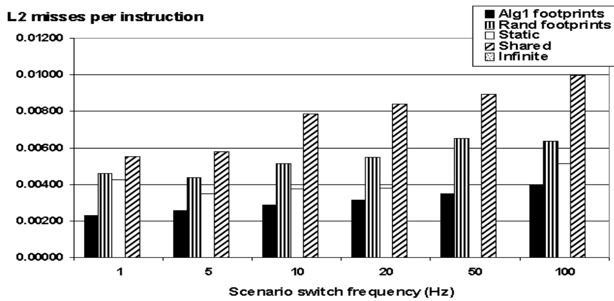

Figure 6. Performance: L2 MPI

When compared with a statically partitioned cache our method exhibits, on average, 25% less MPI, 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 100Hz 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 1Hz 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 (in practice this rate is likely to be even lower than one switch every second), can be beneficial for performance.

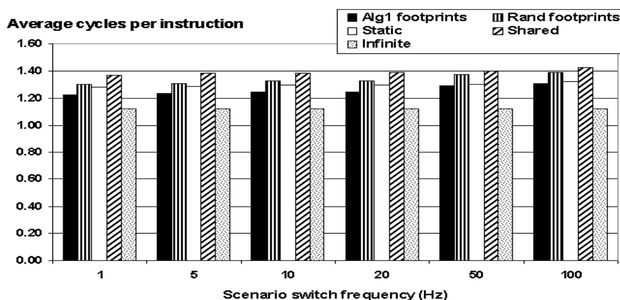

Figure 7. Performance: CPI

When looking solely at the proposed repartitioning we can notice that the MPI increases with 40% when the scenario switching frequency is varied from 1Hz to 100Hz. As a result the CPI increases with 6% in this switching range. However, for realistic switching ranges (over 10Hz) the difference in MPI is on average 18% and in CPI is 1%, suggesting that in such cases the flushing penalty is negligible.

5 Conclusions

In this paper we proposed a dynamic cache management method that enhances compositionality for multimedia applications with multiple utilization scenarios. Our method determines at design time the cache footprint of each task, such that the critical tasks are guaranteed to be undisturbed, and the repartitioning overhead is minimized. Moreover at run time it further decreases the repartitioning penalty. we investigated the compositionality and the performance induced by the L2 repartitioning over a wide range of scenario switching frequency (100Hz to 1Hz), on a CAKE multiprocessor with 4 cores. The workload consisted of six applications formed by various task from the MediaBench suite augmented with an H.264 algorithm. 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 achievable by using an infinite L2 cache. Therefore, despite the involved cache flushing, the repartitioned L2 enables high compositionality and performs better than the shared cache.

References

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

[2] L. Chunho *et al.* Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. *Proc., MICRO*, 1997.

[3] H. Dybdahl and P. Stenstrom. An adaptive shared/private nuca cache partitioning scheme for chip multiprocessors. *Proc. of HPCA*, 2007.

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

[5] D. B. Kirk. Smart (strategic memory allocation for real-time) cache design. *Proc., RTS*, 1989.

[6] J. Liedtke *et al.* Os-controlled cache predictability for real-time systems. *Proceedings, RTAS*, June 1997.

[7] A. Molnos *et al.* Compositional memory systems for multimedia communicating tasks. *Proc., DATE*, 2005.

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

[9] H. Muller *et al.* Caches with compositional performance. *Proc., Embedded Proc. Design Challenges*, 2002.

[10] B. A. Nayfeh and K. Olukotun. Exploring the design space for a shared-cache multiprocessor. *Proc. ISCA*, 1994.

[11] A. Settle *et al.* A dynamically reconfigurable cache for multithreaded processors. *In Jour. of Emb. Comp.*, 2006.

[12] G. E. Suh *et al.* Dynamic partitioning of shared cache memory. *The Jour. of Supercomp.*, 2004.

[13] Y. Tan and V. Mooney. A prioritized cache for multi-tasking real-time systems. *Proc., SASIMI*, 2003.

[14] J. T. van Eijndhoven *et al. Chapter 4 of Dynamic and robust streaming between connected CE-devices.* Kluwer, 2005.