Static Cache Partitioning Robustness Analysis for Embedded On-Chip Multi-Processors

Anca M. Molnos[1], Sorin D. Cotofana[2],
Marc J.M. Heijligers[1], Jos T.J. van Eijndhoven[1]

[1] NXP Semiconductors, HTC 31, Eindhoven, The Netherlands
[2] Technical University of Delft, Mekelweg 4, Delft, The Netherlands

Abstract. In this paper we propose a method to analyze the robustness of multi-tasking media applications when mapped on an on-chip multi-processor platform. We assume a multiprocessor structure which embeds a cache hierarchy with two levels: an L1 that each processor may have and an L2 shared among the processors. To enable compositionality, i.e, to be able to evaluate the system performance out of the individual tasks performance, this shared L2 is partitioned per task basis. In this paper we first introduce two metrics to quantify the robustness. The internal robustness is estimated by a sensitivity function which measures the performance variations induced by the inter-task cache interference. The external robustness is quantified by a stability function which reflects the variations induced by different input data on the partitioned L2 behavior. Subsequently, we exercise our method on a set of multi-media applications running on a CAKE multi-processor platform. Our experiments indicate that, if the cache is partitioned, the sensitivity is on average 4%. whereas for the shared cache it is 25%. Over the investigated workloads the stability is at least 90% therefore, for the those applications, we can conclude that the static cache partitioning is quite robust to input stimuli.

1 Introduction

State-of-the-art media applications are characterized by high requirements with respect to computation and memory bandwidth. On the computation side, the embedded domain low power and low cost demands make the use of general purpose architectures with clock frequencies in the order of several GHz inappropriate. Instead, on-chip multi-processor architectures are preferred. On the memory side, media applications process large amounts of data residing off-chip. The availability of these data at the right moments in time is critical for the application performance, therefore a common practice is to buffer parts of the data on an on-chip memory.

A possible organization of the on-chip memory which alleviates the data availability problem is based on hierarchical caches. In such a context each and every processor core has associated its private cache memory (called L1 cache in this paper). As these L1 caches cannot provide the required application bandwidth [1], shared level two (L2) caches are used [9], [10]. The advantage of an L2

is that large part of the data is kept on chip, where the access is at least 10 times faster than an off chip access [12]. The disadvantage of such a shared L2 cache is that different tasks may flush each others data out of the cache, leading to an unpredictable number of L2 misses. As a consequence, the system performance cannot any longer be derived from the individual tasks performance (property addressed as compositionality).

For media applications guaranteeing the completions of tasks before their deadlines is of crucial importance. Therefore, predictability and robustness are among the main required properties in this domain. A solution for the predictability problem is to use static partitioning of the cache as proposed in [14]. In this approach, the compositionality is induced by allocating parts of the L2 cache, exclusively, to each individual task in the application. However, the compositionality is not 100% ensured because the L1 cache is assumed to be private to each and every task during its execution and only the L2 is partitioned. Thus, in order to guarantee performance, one should be able to estimate the variations induced by the L1 inter-task sharing.

Moreover, static cache partitioning is utilized, thus the application may use only one partitioning ratio during its entire execution. This cache partitioning ratio is computed utilizing the application's statistics for a given input data set [13]. However, during the application execution different other input data might have to be processed. It is quite probable that for these new data sets the partitioning ratio for which the application has its best performance is different than the one which is in use. To be able to guarantee performance, the designer should be able to estimate these deviations too.

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

In this paper we propose an approach to assess the robustness of an application running on a multi-processor system with statically partitioned L2. The present article is an extension of the work in [17]. As previously mentioned, for this type of systems the internal robustness is determined by inter-task interference in the L1 cache. This interference strongly depends on the task switching rate. To estimate the internal robustness we introduce a sensitivity metric which reflects the variation in L2 misses number for different task switching rates. To assess the external robustness, we introduce the stability metric. It measures the performance deviations for the case when the application processes another input data set than the one utilized to determine the static partitioning ratio. An application is considered to be stable if its number of misses obtained with a certain input data is close to the least number of misses possible for that input data.

To demonstrate our approach we analyze two types of parallel applications: (1) applications consisting of communicating tasks and (2) applications consisting of independent tasks. From the first category we exercise two applications: a picture-in-picture (PiPTV) video decoder and an H.264 decoder. From the second category we analyzed six applications composed by different multimedia tasks. We utilize a CAKE multi-processor instance [10] as simulation platform. For these applications, we evaluate the sensitivity function (internal robustness) and the stability function (external robustness). Our experiments indicate that, if the cache is partitioned, the sensitivity is on average 4% whereas if the cache is shared the sensitivity is 6 times larger. Thus, as expected, cache partitioning drastically reduces the inter-tasks conflicts. Most important, for our applications, this small percentage of variations suggests that partitioning the L2 is enough to achieve compositionality in a large degree. The variations induced in the L2 behavior by various input data sets are at most 10% over all the application range that we tried. This accounts for an average stability of 92%, therefore, for the investigated applications, we can conclude that the static cache partitioning is quite robust with respect to input stimuli variations.

The remainder of the paper is organized as follows. Background information over the considered multi-processor platform and the cache partitioning method is introduced in Sect. 2. The robustness evaluation method is described in Sect. 3. Sect. 4 presents practical experiments and results, and Sect. 5 concludes the paper.

2 Background

This section introduces briefly the targeted system and the application model, and then details our task centric cache management scheme.

2.1 Target Architecture

The envisaged multi-processor architecture consists of a homogeneous network of computing tiles on a chip [10]. Each tile contains a number of CPUs, a router (for out of tile communication), and memory banks. The processors are connected to memory by a fast, high-bandwidth interconnection network. Each of the processor cores has its own L1 cache. The on-tile memory is actually used as a large, unified L2 cache, shared between processors, facilitating a fast access to the main memory which resides outside the chip. In case a task doesn't find its required data in the corresponding L1, a coherence protocol is executed to determine if the data are located in another processor L1 cache. If the data are not present in none of the L1 caches, the L2 is accessed. In this paper we use one tile of the multi-processor like the one depicted in Fig. 1.

An application executed on this architecture consists of multiple tasks. These tasks may exchange data among each other, or they may be independent. If inter-task communication is present, we assume that it is performed through the memory hierarchy, thus through the shared L2.

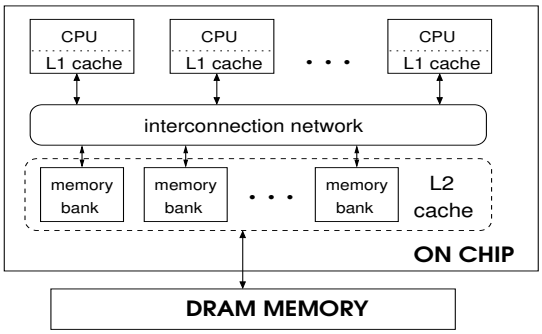

Fig. 1. Multi-processor target architecture.

Each task can be regarded as a process consuming input data and producing output data. For an application formed by communicating tasks, these tasks are naturally synchronized based on data availability. In this case a task temporarily stops its execution (is swapped out) in two cases: (1) when task's input data buffers are empty or its output buffers are full, (2) when an interrupt occurs. In the case an application is formed by independent tasks, these tasks may stop their execution when the task scheduler policy dictates. Between two executions of the same task, a processor can execute other tasks. Moreover, in order to support a natural load balancing, the tasks may freely migrate from one processor to another, depending on the processors availability.

2.2 Cache Partitioning

We assume a conventional cache to be a rectangular array of memory elements arranged in "sets" (rows) and "ways" (columns). The accessed address is logically split in three fields: *tag*, *index*, and *offset*. The *offset* part of the address identifies the required data word inside a cache line. The set where a data item can be placed is uniquely identified by the *index* part of the address. Inside that set, the data may reside in one of the ways. In case some data item is required, all the ways are searched to determine if and in which one of them it is cached. In a traditional cache, neither the index addressing, nor the replacement policy are aware of the internal, task-based, structure of the application. This unawareness may cause unpredictable inter-task misses which should be avoided in order to ensure compositionality.

We assume that in a CAKE-like organization, the L2 is likely to be largely affected by inter-task cache conflicts, as it is shared among the processors. Although the L1s are also shared among task, this sharing is different than the L2 sharing. Tasks can successively execute on the same processor, but on a given moment in time only one task can evict data out of the processor L1. Thus we assume the L1 as private to each task during its execution. As a results, our we focus on isolating tasks in the L2 (assign a L2 part to each one of them), such

that their number of misses are independent of each other. For this we utilize a cache partitioning scheme.

Based on the conventional cache organization, there are two main natural partitioning manners: (1) based on associativity and (2) based on sets. In the remainder of this subsection we describe in detail these main options, their potential implementation in the context of the CAKE multiprocessor architecture and in the end of this section we detail the manner we use them to enhance compositionality.

Associativity Based Partitioning. The associativity based partitioning scheme is depicted in Figure 2. As one can observe, each and every task gets a number of ways from every set of the cache. In case the required data item is present in the cache, it is accessed, just like in a conventional cache. However, in case of a miss, when a cache line has to be replaced, one task can flush out only its own cache ways. In this manner, different tasks do not interfere unpredictably.

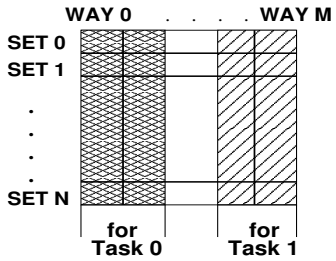

Fig. 2. Associativity based cache partitioning (logic organization).

This type of partitioning is implemented by changing the cache replacement policy as suggested in [2]. This require a small table that specifies which task owns which cache ways, and some extra logic to restrict the victim lines that can be flushed. This logic is not on the critical path, as the line to be victimize does not have to be known before the data are actually loaded from a lower memory level. On our CAKE platform, loading an L2 line from the main memory takes at least 90 cycles, thus we can consider that there is no time penalty involved in associativity based partitioning. From the area point of view, all the necessary hardware represents a negligible fraction of the size of a L2 cache. This negligible penalty, together with the fact that the implementation doesn't require modifications in the structure of the cache or in the addressing mode [2], leads to a common use of variations of this partitioning type [7] [11] [15] for the purpose of reducing the number of misses and speeding up the application.

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

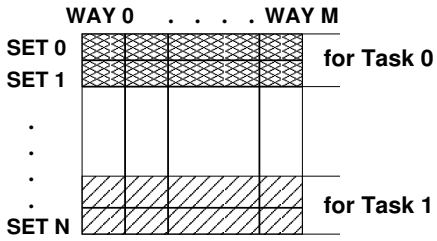

Fig. 3. Set based cache partitioning (logic organization).

extra comparator on the critical path [12]. Thus the reason for supporting just few ways is that extra circuitry involved in implementing associativity slows down the cache and burns a lot of power at each lookup. In media applications there is a trend in adding new features, so increasing the number of tasks. Consequently, 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.

Set Based Partitioning. The set based partitioning scheme is illustrated in Figure 3. In this case, each and every task gets a different amount of sets from the cache. As already mentioned, in a conventional set associative cache organization the address splits into three parts: tag, index, and offset. Set based partitioning implies that the addresses a task accesses may have only some restricted indexes, pointing to the task's cache sets. This is equivalent with an address space partitioning. To our knowledge, there are two previous approaches to implement this address space partitioning. One implements the partitioning at compiler and linker level [8] and the other at operating system level [4]. In the scheme proposed in [8] the compiler and the linker allocate variables and instructions addresses such that the cache partitioning is achieved. In our case the platforms may contain standard processor cores, thus the compilers are developed by external parties. A platform specific change of the compiler would be costly or time consuming. The cache partitioning method controlled by the operating system proposed in [4] has also drawbacks as it is limited to physically indexed caches and requires a virtual memory model. In our approach, we would like to support all types of caches on platforms with or without memory paging. Consequently, none of the existing method are suitable for our purpose, thus in the following we propose a new technique to implement set based cache partitioning.

We achieve the cache partitioning through a level of indirection, without interfering with the memory space. This is somewhat similar with the mechanism in [4], but the address translation is not at memory page level, but directly at cache level. In this manner there is no restriction in the type of cache supported, nor in the underlying memory model. Our scheme modifies the index bits of an address into new index bits, before cache lookup (Fig. 4), taking into account who initiates the access. The purpose of the index translation is to send all the

access of a task T_i, and only the accesses of task T_i, in a cache region decided at design time.

To avoid expensive index calculation, the partition sizes are limited to power of two number or 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 representative bits of idx_A (instead of doing modulo with the cache size 2^C we do only 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 didn't have the same old index might end up having the same new index. In this case the system is not able to distinguishes among such two addresses, leading to data corruption. To prevent data corruption, the index bits changed by the translation process still have to identify somehow the associated memory access. The easiest way to achieve this is to augment the tag part of the address with those changed index bits. For our example, task T_i has 2^k cache sets thus the $C-k$ most representative are changed, and have to be included in the tag. Because it is not beneficial to have a tag with variable length (k varies with the task's allocated cache size) we choose to augment the tag with all index bits. In this case, for instance, for 2MBytes 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 described in the beginning of this section. In case a task doesn't 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 execution of the coherence protocol is always launched before of an L2 access, and takes few cycles. Consequently, the index translation for L2 accesses can be performed in parallel with the cache coherence resulting in no additional delay penalty associated to the extra index translation.

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

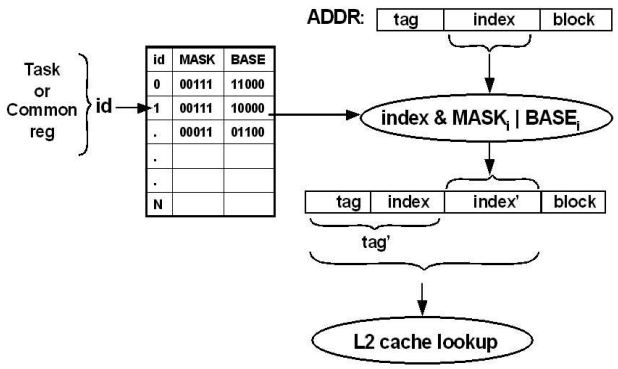

Fig. 4. Set based cache partitioning (implementation).

2.3 Task Centric Cache Management

As aforementioned, we consider that an application can be formed by communicating or independent tasks. In the following we briefly present our cache management strategy for both these types of applications.

In the case of independent tasks compositionality is realized by assigning a part of cache to each task. The experiments in [6] suggest that for static partitioning, the set based strategy performs better that the associativity based one. The reason for this is that associativity based cache partitioning is decreasing the number of ways a task can use. It is known that, having a fixed cache size, a cache organization with a large associativity (and a small number of sets) performs most of the times better than one with less associativity (but more sets) [12, ?]. In addition, as we already mentioned, the set based method offers, for the same cache dimensions, more allocable cache units than the associativity based one, as in a state-of-the-art cache the number of sets is few orders of magnitude larger than the number of ways. Therefore, in the case of independent task we opt for set based partitioning.

In the case of application consisting of communicating tasks, the problem that arise is where to cache the shared data. To solve this problem we proposed in [14] a mixed set and associativity based partitioning. First, each task and each inter-task communication buffer gets an exclusive part of the cache sets. Second, inside the cache sets of a communication buffer each task accessing it gets a number of ways. In this manner tasks may compositionally share data or instructions. As there is no principial difference between sharing data or instructions, we use the term "common region" to denote both these shared parts.

The partitioning ratio is determined such that the overall application number of misses is minimized. Let us assume that in the general case an application A is composed out of N tasks, $T = \{T_i\}_{(i=1,N)}$, and M common regions $CR = \{CR_j\}_{(j=1,M)}$ (for the particular case of an application composed out of independent tasks $M = 0$). The process of finding this optimized ratio requires first an information gathering phase during which every task T_i is individually

simulated having different amounts of cache. Subsequently, the best partitioning ratio is computed such that the sum of all task misses is minimized, under the constraint that all allocated cache cannot be larger than the available cache. This best partitioning ratio BPR is a set of cache sizes $\{c_i\}_{(i=1,N+M)}$, where c_i is the cache allocated to task T_i, or to common region CR_j (we consider that the indexes j of the common regions actually run from $N+1$ to $N+M$). Using these notation, in the following section we introduce the two metrics for assessing the application robustness.

3 Robustness Evaluation Method

This section presents the proposed approach to assess the robustness of an application running on a multi-processor as the one described in Subsection 2.1. We consider two aspects of robustness: (1) internal robustness defined as the sensitivity of the L2 misses of a task on the other tasks' behavior, (2) external robustness defined as the variations induced in the L2 behavior by various input data sets.

3.1 Internal Robustness

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

For internal robustness analysis we propose to use the L2 sensitivity function. In order to define it, let us assume that the application is composed out of N tasks, $T = \{T_i\}_{(i=1,N)}$ and that $SWR = \{swr_r\}_{(r=1,R)}$ is the set of investigated task switching rates. The number of L2 misses of task T_i depends on T_i's allocated cache size c_i, and on the task switching rate swr_r. We denote these T_i's misses with $miss_i(c_i, swr_r)$. The L2 sensitivity corresponding to a task T_i is defined as being the maximum difference in the number of L2 misses among the investigated task switching rates, when a given L2 cache size c_i is allocated to T_i. To give an idea about the impact of this variation on the application performance, we define the task sensitivity relative to the number of misses obtained when the tasks switch at a reference rate, swr:

$$sens_i(c_i) = \frac{\left| \max\limits_{SWR}\{miss_i(c_i, swr_r)\} - \min\limits_{SWR}\{miss_i(c_i, swr_r)\} \right|}{\sum\limits_{i=1}^{N} miss_i(c_i, swr)} \times 100\%. \quad (1)$$

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

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

$$sens_A = \max_{T_i \in T}\{sens_i(c_i)\}. \quad (2)$$

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

3.2 External Robustness

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

Let us assume that the investigated application has three tasks ($N = 3$) and two relevant sets of input data in_1 and in_2 are considered in the cache partitioning process. Let us assume that when the application uses in_1 (in_2) as input data its best performance is achieved if tasks have as partitioning ratio $BPR_1 = \{c_1^1, c_2^1, c_3^1\}$ ($BPR_2 = \{c_1^2, c_2^2, c_3^2\}$), as depicted in Fig. 5. BPR_1 and BPR_2 are calculated such that the application's L2 misses is minimum, under the constraint that the allocated cache is smaller that the available cache (14 units in our case).

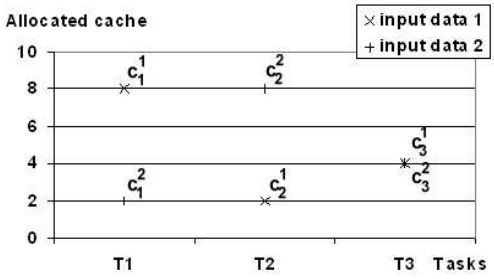

Fig. 5. Example: Partitioning ratios corresponding to two input data.

It can be observed that the best partitioning ratio BPR_1 and BPR_2 are different. When using static cache partitioning the application may use just one single partitioning ratio, $BPR = \{c_1, c_2, c_3\}$. This ratio can be BPR_1, BPR_2, or any compromise between those two. For instance any partition with $c_1 \in [\min(c_1^1, c_1^2), \max(c_1^1, c_1^2)]$, $c_2 \in [\min(c_2^1, c_2^2), \max(c_2^1, c_2^2)]$, and $c_3 = c_3^1 = c_3^2$ can be utilized.

If, for example, BPR_1 is not used as the partitioning ratio, in case the application is processing in_1 as input data, its performance is deviating from the best achievable one. In this case it is of interest to estimate an upper bound of the potential performance degradation. For this purpose, we calculate the worst partitioning ratio, $\overline{BPR_1} = \{\overline{c_1^1}, \overline{c_2^1}, \overline{c_3^1}\}$, with $\overline{c_1^1}, \overline{c_2^1}$, and $\overline{c_3^1}$) bounded by BPR_1 and BPR_2. $\overline{BPR_1}$ is determined utilizing the same optimization method as for BPR_1, but with the constraints that $\overline{c_1^1} \in [\min(c_1^1, c_1^2), \max(c_1^1, c_1^2)]$, $\overline{c_2^1} \in [\min(c_2^1, c_2^2), \max(c_2^1, c_2^2)]$, and $\overline{c_3^1} = c_3^1 = c_3^2$. Because we want to estimate the worst performance, the number of misses is maximized instead of minimized.

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

$$M_1 = miss_1(c_1^1, in_1) + miss_2(c_2^1, in_1) + miss_3(c_3^1, in_1). \quad (3)$$

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

$$\overline{M_1} = miss_1(\overline{c_1^1}, in_1) + miss_2(\overline{c_2^1}, in_1) + miss_3(\overline{c_3^1}, in_1). \quad (4)$$

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

In media applications, time deadlines are imposed for processing a number of data units, for example a video decoder might have to decode 25 frames in a second. Therefore, it is also interesting to evaluate the variations in L2 behavior caused by different data units belonging to the same input stream. This means that, for instance, input data in_1 may be the first frame of a video stream and

in_2 may be the next frame of the same video stream. Such a stability evaluation is useful because it gives a bound of the dynamic behavior inside the same input stream.

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

$$M_l = \sum_{i=1}^{N} miss_i(c_i^l, in_l). \tag{5}$$

For every input data $in_l \in IN$ the best partitioning ratio BPR_l is the set of tasks' allocated cache sizes $\{c_1^l, c_2^l, ..., c_N^l\}$. As previously mentioned, it is possible that the best partitioning ratio BPR_l differ among each other. The final partitioning ratio, $BPR = \{c_1, c_2, ..., c_N\}$ can be BPR_1, BPR_2, ... , BPR_L or any compromise among them, that respects the following condition:

$$c_i \in \left[\min_{IN}\{c_i^l\}, \max_{IN}\{c_i^l\}\right]. \tag{6}$$

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

$$\overline{M_l} = \sum_{i=1}^{N} miss_i(\overline{c_i^l}, in_l). \tag{7}$$

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

$$stab_l = \frac{M_l}{\overline{M_l}} \times 100\%. \tag{8}$$

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

$$stab_A = \min_{IN}\{stab_l\}. \tag{9}$$

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

3.3 Robustness Considerations for Dynamic Cache Repartitioning

An good overview of dynamic cache repartitioning schemes is given in [11]. Similar to the static partitioning case, there are mainly two types of dynamic cache repartitioning. The first is the associativity based repartitioning. The number of cache ways (cache organization) limits the granularity of this partitioning type. Repartitioning is cheap because data correctness can be preserved without flushing the cache. The second is the "set based" repartitioning. Typically in a cache there are more sets than ways, thus this method can potentially offer finer partitioning granularity. However, at repartitioning data correctness cannot be preserved without flushing parts of the cache. This makes this second type of cache repartitioning more expensive than the first type.

The existing dynamic cache repartitioning scheme are associativity based [15] [16]. In these schemes the task that have either high priority [16] or large cache needs [15] dynamically "steals" cache ways from the other tasks. The purpose is to increase the performance of high priority tasks [16] or to improve the overall hit rate [15].

An allocation scheme in which a task will be granted all the requested cache can lead to cache "starvation" of some of the tasks. For example, a repartitioning strategy that attempts to improve the overall hit rate will eventually give a large part of cache to an erroneous task asking for it. Given this fact, the system will fail. Therefore, in a robust system, the cache repartitioning cannot be done fully at tasks requests, like in the existing approaches. The cache manager should have a global view on the application's tasks and their allocated cache, to prevent starvation and system failures. Our future work will include robust dynamic cache repartitioning strategies.

4 Experimental Results

For our experiments we use a CAKE multi-processor platform [10] with 4 Trimedia processor cores and a 4 ways associative L2 cache of various sizes, depending on the application. Each and every Trimedia processor core has separate instructions and data L1 caches. The shared L2 cache is unified (it contains both data and instructions). To enhance compositionality, we use the mixed L2 partitioning previously presented in Sect. 2.3. The experimental workload consists of two application types: (1) applications composed out of communicating tasks and (2) application composed out of independent tasks. In the following we introduce these applications.

From the first category the applications we consider are two video decoders (an H.264 decoder and a picture-in-picture-TV decoder), each of which consisting of several communicating tasks. The H.264 decoder consists of 15 tasks [18], as follows: first an entropy decoder task processes the input stream and passes the data via a data scheduler to a set of transform decoders and loop filters tasks doing inverse quantization, transformation, prediction and deblocking on different parts of the image. The PiPTV application decodes two different video streams and outputs a raw pictures stream containing both video stream images, scaled with a given factor. This application consists of the following tasks: video demultiplexing of transport stream, two MPEG2 decoders (every one having multiple tasks [19]), two video scalers, video multiplexing the two images, and output. Both these applications are described in YAPI [20].

In order to build applications formed by independent tasks, we use various multimedia programs, some of which derive from the MediaBench benchmark [5]. From this collection of programs we pruned out the ones that are relatively small and not memory intensive. Moreover, in order to make the benchmark more representative for emerging technologies, we augmented the MediaBench suite with two H.264 video processing programs, an encoder and a decoder. For clarity sake, we emphasize the fact that all these programs are sequential and different than the H.264 decoder or the MPEG2 decoders introduced in the previous paragraph. In the experimental framework, an application is formed by a collection four such programs, each of which representing a task. Table 1 presents the set of 9 tasks exercised. All of these are reasonably memory intensive workloads. Using different combinations of these 9 tasks, we build 6 different applications ($A_1, A_2, ..., A_6$).

Table 1. Media workloads.

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

In the remainder of this section the robustness assessment methods we introduced in Sect. 3 are applied. The results obtained for the case of the partitioned cache are compared with the ones for the shared cache. To our knowledge, no cache related robustness investigation method exists in the literature, therefore we cannot compare our proposal with previous work.

4.1 Internal Robustness

As aforementioned, the applications consisting of communicating tasks are described in YAPI, thus the data exchange and synchronization among the tasks is done through blocking FIFOs. A task is blocked (and consequently its processor switches to other task) when it has no available input data or output buffer space. On our experimental platform, for the purpose of our investigations, we induce higher task switching rate by shrinking the FIFOs sizes. For FIFOs larger than a certain size the task switching rate does not decrease anymore because a value intrinsic to the application is reached. We consider this lowest value as the reference task switching rate, as defined in the Sect. 3.1. In our case, both applications have the least number of misses for the lowest task switching rate. The internal robustness is relative to this number of misses, therefore the presented results reflect the largest deviations.

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

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

For both application types, the L2 sensitivity of tasks is compared for the partitioned and the shared cache case (Fig. 6). In Fig. 6 are depicted only the tasks that have the sensitivity larger than 2% in the partitioned cache case or larger than 20% in the shared cache case. In this figure it can be observed that, in general, the shared L2 is more sensitive than the partitioned one or their sensitivities are pretty close. Among the tasks that are not depicted in Fig. 6, there are few for which the sensitivity of the partitioned L2 cache is larger than the one of the shared cache. However, for those few tasks, the sensitivity is smaller than 0.5%, so they do not influence the general observed trend i.e. the shared cache is more sensitive that the partitioned one. Fig. 7 presents the application sensitivity for our eight case. Over all the applications, the shared cache is on average 6 times more sensitive to task switching than the partitioned one. The largest sensitivity was observed at the applications $H.264$ and $PiPTV$ for the case of partitioned and shared cache, respectively. For a partitioned cache, over the investigated task switching range, the application sensitivity as defined in

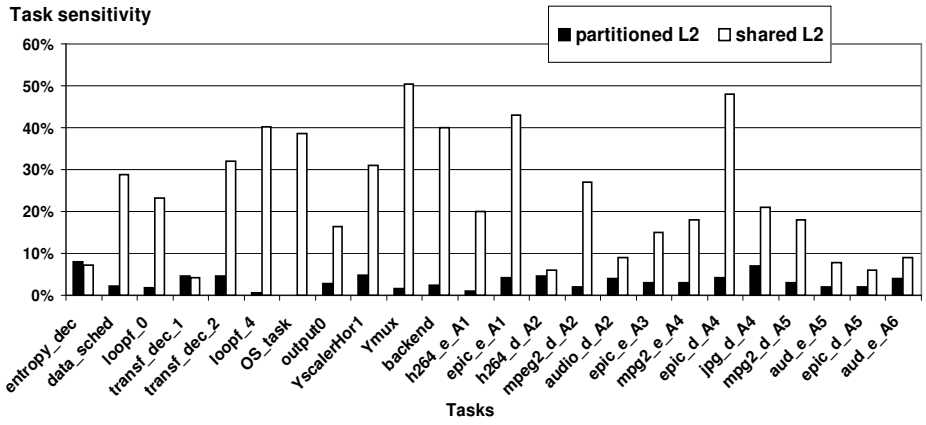

Fig. 6. Tasks sensitivity: shared vs. partitioned cache.

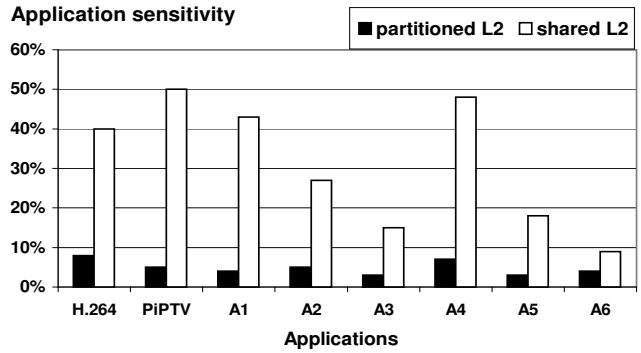

Fig. 7. Application sensitivity: shared vs. partitioned cache.

Sect. 3.1 is at most 8%. These results suggest that, for the analyzed applications, partitioning the L2 is enough to achieve compositionality to a large extent.

4.2 External Robustness

As detailed in section 3.2, the best cache partitioning ratio of an application varies with its tasks input data. In order to quantify the differences among the best cache partitioning ratio, we use the maximum variation of the L2 size allocated to a task, across different input streams. Table 2 depicts (for each application) the maximum variation of the L2 size allocated to a task, across different input streams. The values in Table 2 are relative to the total cache size available to each application. In general we found that the differences among the best partitioned ratio corresponding to different input data are relatively small. As one can see in Table 2, over the 8 applications that we exercised, the cache of a task varies at maximum with 20% from the total cache size.

Table 2. Maximum variation in the L2 size allocated to a task.

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

For some input data, the partitioning ratio is non-optimal and this induces a performance degradation. To quantify this degradation, in Sect. 3.2 we introduced the stability metric. In Table 3 the stabilities corresponding to each application are illustrated. For all the eight applications we investigated four different input data streams (these streams are the one required by each application task, as some tasks decode video, some process audio, etc.). We would like to mention that the set of input data corresponding to a task has the same size and the same "quality" level for each experiment. In this section we do not investigate the effects of things like enlarging the resolution or the scaling factor of a video stream, or changing the encoding quality of an image. These issues are subject to future work.

Table 3. Application stability for different input data.

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

Fig. 8 presents, for each of the eight applications, the minimum stability over the set of four input streams. We observe that the minimum stability of each application is pretty high, ranging over the eight applications from 90% to 96%, with an average of 92%. Taking these facts into account, we can conclude that all the eight applications are quite robust to input stimuli in the presence of static cache partitioning. A stability comparison between the shared and the partitioned cache is not possible because the stability, as defined in Sect. 3.2, is linked to the partitioned ratio, thus it cannot be computed for the shared cache scenario.

5 Conclusions

In this paper we proposed a method to analyze the static cache partitioning robustness of an application mapped on an on-chip embedded multi-processor. In this context we considered a memory organization which has two levels of cache:

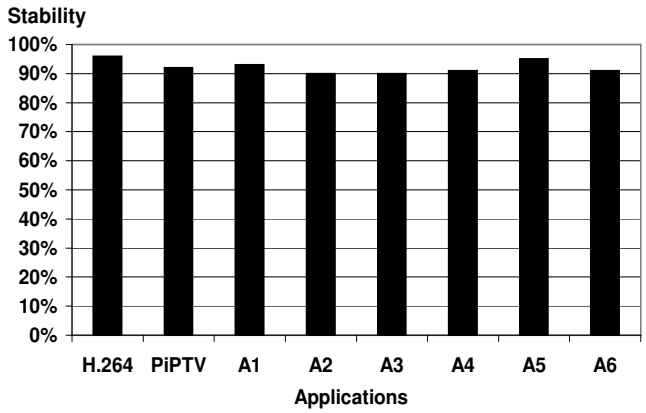

Fig. 8. Minimum application stability.

(1) L1, private to every processor and (2) L2, shared between the processors, but partitionable per task basis. For applications executed on this multi-processor, two types of robustness are discussed: internal (determined by inter-task interference in the not-partitioned L1 cache) and external (determined by the variations of the L2 behavior due to various input data sets). For both types of robustness we introduced quantification metrics. For internal robustness we defined the sensitivity function which measures the deviation of L2 misses caused by the L1 variations over a spectrum of task switching rates. To assess external robustness we introduced the stability function which measures the performance deviation for the case the application processes another input data set than the one utilized to determine the static L2 partitioning ratio.

To demonstrate our approach we analyzed two types of parallel applications: (1) applications consisting of communicating tasks and (2) applications consisting of independent tasks. In the first category we analyzed two applications: a picture-in-picture video decoder and an H.264 decoder. In the second category we analyzed six applications each of which composed by different multimedia tasks. These tasks were chosen from the MediaBench suite, augmented with two more programs, an H.264 encoder and an H.264 decoder. The simulation platform is a CAKE multi-processor instance. Concerning the internal robustness, if the cache is partitioned, the application sensitivity is at most 8% with an average of 4%. This small sensitivity suggests that partitioning the L2 is enough to achieve compositionality in a large degree, for these applications. Comparing the internal robustness of the shared and partitioned cache cases, we found that the shared cache is on average 6 times more sensitive than the partitioned one. Moreover, the large difference among the shared cache and the partitioned cache sensitivity is an interesting fact on itself. It suggests that the optimizations processes for L1 and L2 caches can be decoupled if the L2 is managed on a task centric manner. Concerning the external robustness, the variations

induced in the L2 behavior by various input data sets are at most 10% over all the application range that we tried. This accounts for an average stability of 92%, therefore, for the investigated applications, we can conclude that the static cache partitioning is quite robust with respect to input stimuli variations.

References

1. A. Stevens, "Level 2 Cache for High-performance ARM Core-based SoC Systems", ARM white paper, 2004
2. D. T. Chiou, "Extending the Reach of Microprocessors: Column and Curious Caching", PhD thesis Department of EECS, MIT, Cambridge, MA, 1999
3. Allan Hartstein, Viji Srinivasan, Thomas R. Puzak, Philip G. Emma, "Cache miss behavior: is it sqrt(2)?", Conf. Computing Frontiers, pages 313-320, 2006
4. Jochen Liedtke, Hermann Härtig, Michael Hohmuth, "OS-Controlled Cache PRedictability for Real-Time Systems", 3rd IEEE Real-Time Technology and Applications Symposium, 1997
5. L. Chunho, M. Potkonjak, W.H. Mangione-Smith. "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems" In *International Symposium on Microarchitecture*, 1997.
6. A. M. Molnos, Marc J.M. Heijligers, Sorin D. Cotofana, Jos T.J. van Eijndhoven, "Compositional Memory Systems for Data Intensive Applications", Proceedings, Design, Automation and Test in Europe, 2004
7. Harald S. Stone, John Truek, Joel L. Wolf, "Optimal Partitioning of Cache Memory", IEEE Transactions on computers, volume 41, number 9, pages 1054-1068, 1992
8. F. Mueller, "Compiler Support for Software-Based Cache Partitioning", In *ACM SIGPLAN Notice*, 1995.
9. B.A. Nayfeh and K. Olukotun, "Exploring the Design Space for a Shared-Cache Multiprocessor", In Proceedings, ISCA, pages 166-175, 1994
10. J.T.J. van Eijndhoven, J. Hoogerbrugge, M.N. Jayram, P. Stravers, and A. Terechko, "Cache-Coherent Heterogeneous Multiprocessing as Basis for Streaming Applications", In "Dynamic and robust streaming between connected CE-devices", Kluwer Academic Publishers, 2005
11. P. Ranganathan, S. Adve, and N.P. Jouppi, "Reconfigurable caches and their application to media processing", In Proceedings, 27th Annual International Symposium on Computer Architecture, pages 214-224, 2000
12. J.L. Hennesy and D.A. Patterson, "Computer Architecture: A Quantitative Approach", Morgan Kaufmann Publishers, 2003
13. A.M. Molnos, M.J.M. Heijligers, S.D. Cotofana, and J.T.J. van Eijndhoven, "Compositional memory systems for multimedia communicating tasks", In Proceedings, Design, Automation and Test in Europe, pages 932-937, 2005
14. A.M. Molnos, M.J.M. Heijligers, S.D. Cotofana, and J.T.J. van Eijndhoven, "Compositional, efficient caches for a chip multi-processor", In Proceedings, Design, Automation and Test in Europe, to appear in 2006
15. G.E. Suh, L. Rudolph, and S. Devadas, "Dynamic Partitioning of Shared Cache Memory", The Journal of Supercomputing, volume 28, number 1, pages 7-26, 2004
16. Y. Tan and V.J. Mooney, "A Prioritized Cache for Multi-tasking Real-Time Systems", In Proceedings of the 11th Workshop on Synthesis And System Integration of Mixed Information Technologies, pages 168-175, 2003

17. A.M. Molnos, S.D. Cotofana, M.J.M. Heijligers, and J.T.J. van Eijndhoven, "Static cache partitioning robustness analysis for embedded on-chip multi-processors", In Proceeding of the ACM International Conference on Computing Frontiers, 2006
18. E.B. van der Tol, E.G. Jaspers, and R.H. Gelderblom, "Mapping of H.264 decoding on a multiprocessor architecture", In Proceedings, SPIE Conference on Image and Video Communications and Processing, 2003
19. P. van der Wolf, P. Lieverse, M. Goel, D. La Hei, K.A. Vissers "An MPEG-2 Decoder Case Study as a Driver for a System Level Design Methodology", In Proceedings, 7th International Workshop on Hardware/Software Co-Design, pages 33-37, 1999
20. E. A. de Kock, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, P. Lieverse, K. A. Vissers, and G. Essink "YAPI: application modeling for signal processing systems", In Proceedings, 37th conference on Design Automation, pages 402-405, 2000