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

A.M. Molnos, S.D. Cotofana
Computer Engineering Laboratory
Delft University of Technology
Mekelweg 4, 2426 CD, The Netherlands
(A.M.Molnos,S.D.Cotofana)@tudelft.nl

M.J.M. Heijligers, J.T.J. van Eijndhoven
Philips Research Eindhoven
HTC 5, 5656 AE, The Netherlands
marc.heijligers@philips.com,
jos.van.eijndhoven@philips.com

ABSTRACT

In this paper we analyze the robustness of multi-tasking applications when mapped on an on-chip multiprocessor platform. We assume a multiprocessor structure which embeds a hierarchical cache organization with two levels. The first one is private to the processor cores while the second one is shared among the processors. To enable compositionality, i.e, to be able to evaluate the system's performance out of the individual task's performance, the second level of cache (L2) is partitioned per task basis. Two robustness aspects are relevant in this context: internal (performance deviations are caused by the tasks comprising the application) and external (performance variations are caused by external stimuli). First we 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 two applications (H.264 and picture-in-picture TV) running on a CAKE multi-processor platform. Our experiments indicate that, if the cache is partitioned, the sensitivity is 8% and 5% for the H.264 and PiPTV, respectively. For the shared cache scenario the sensitivity is 40% and 50% for the H.264 and PiPTV, respectively. The variations induced in the L2 behavior by various input data sets are at most 4% for the PiPTV application, respectively 9% for the H.264 decoder. This accounts for a stability of at least 96%, respectively 91%, therefore, for the investigated applications, we can conclude that the static cache partitioning is quite robust to input stimuli.

Categories and Subject Descriptors

C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems; B.3.2 [**Design Styles**]: Cache memories—*robustness measures*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CF'06, May 3–5, 2006, Ischia, Italy.
Copyright 2006 ACM 1-59593-302-6/06/0005 ...$5.00.

General Terms

Reliability, performance, theory.

Keywords

Robustness, multi-processors, cache partitioning.

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 multiprocessor 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 [2], [3]. 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 [5]. 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's performance cannot any longer be derived from the individual task's 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 [6]. 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 [10]. 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. 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 parallel applications: a picture-in-picture video decoder and a H264 decoder. We utilize a CAKE multi-processor instance [3] as simulation platform, the input stimuli available at [7], and we compare the robustness of the shared and partitioned cache cases. For both 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 8% and 5% for the H.264 and PiPTV, respectively. For the shared cache scenario the sensitivity is 40% and 50% for the H.264 and PiPTV, respectively. Thus comparing the internal robustness of the two cases, one can see that the shared cache is 5 times for the H.264 decoder, respectively 10 times for the PiPTV decoder, more sensitive than the partitioned one. The variations induced in the L2 behavior by various input data sets are at most 4% for the PiPTV application, respectively 9% for the H.264 decoder. This accounts for a stability of at least 96%, respectively 91%, therefore, for the investigated applications, we can conclude that the static cache partitioning is quite robust to input stimuli.

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

2. BACKGROUND

This section introduces the targeted system, the application model, and the cache management scheme.

2.1 Target architecture

The envisaged multi-processor architecture consists of a homogeneous network of computing tiles on a chip [3]. 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. Since this L1 cache's latency directly relates to the processor's cycle time, there are very strict timing requirements for this cache. Therefore, the L1 caches are relatively small. 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. If data are not present in the L1 cache the L2 is accessed, so if the L1 performance varies the L2 is impacted. In this paper we use one tile of the multi-processor like the one depicted in Figure 1.

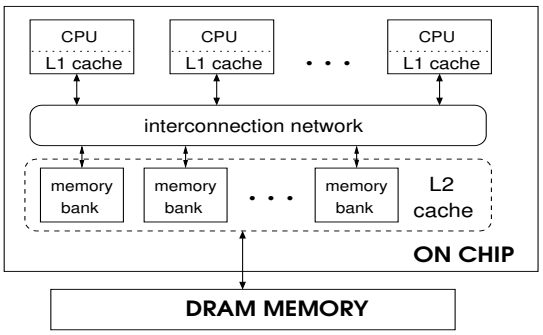

Figure 1: Multi-processor target architecture

The applications executed on this architecture consist of sets of tasks that communicate through the memory hierarchy, thus through the shared L2. Each task can be regarded as a process consuming input data and producing output data. In this way the tasks are naturally synchronized based on data availability. 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. 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

In the considered multi-tasking environment it is possible that two tasks T_i and T_j have their data mapped in the same cache location. Therefore, when T_i's data is loaded into the cache it may flush T_j's data, eventually causing a future T_j miss. This kind of unpredictability constitutes a major problem for real-time applications. Ideally, the designer would like to have a compositional system such that the overall application performance can be predicted based on the performance of its individual tasks. For this purpose exclusive L2 cache parts are statically allocated to tasks and inter-

task communication buffers using the method introduced in [6].

We assume a conventional cache to be a rectangular array of memory elements arranged in "sets" (rows) and "ways" (columns). We perform two partitioning types. 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. The used partitioning ratio is determined such that the overall application number of misses is minimized [10]. Let us assume that the application is composed out of N tasks, $T = \{T_i\}_{(i=1,N)}$. The process of finding this optimized ratio require 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 $\{c_i\}_{(i=1,N)}$, where c_i is the cache allocated to task T_i.

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 executed and on whether other tasks were executed in the mean time on P_k. High task switch rate are likely to pollute L1 caches less at a time, but for many times. Low task switch rate 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 the internal robustness analysis we propose to use the L2 sensitivity function. In order to define it, let us assume that the application is composed out of N tasks, $T = \{T_i\}_{(i=1,N)}$ 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:

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

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

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

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. However, due to typical small sizes, L1 is unsuited for static partitioning. In a multi-processor system, if L1 is statically partitioned the application's tasks should be statically assigned to processors (it makes no sense to allocate cache for a task on a processor where that task might never run). This is not a preferred option because it restricts the run-time processors' load balancing options. For example in a video decoder where all tasks concur for processing frames at a certain rate, restricting run-time load balancing can diminish the performance. In the case that L1 is dynamically partitioned, the application's sensitivity $sens_A$ still cannot be zero because the repartitioning is dictated at run-time, therefore variations may occur.

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 Figure 2. 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 (12 units in our case).

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.

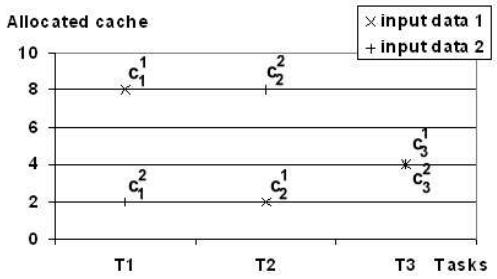

Figure 2: Example: Partitioning ratios corresponding to two input data

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 $(\overline{c_1^1}, \overline{c_2^1}, \overline{c_3^1})$ 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:

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

where $miss_{1,2,3}$ are the number of misses corresponding to 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:

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

The same type of investigation can be done for in_2 also and the values $\frac{\overline{M_1}}{M_1}$ and $\frac{\overline{M_2}}{M_2}$ 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 the same input stimuli. This means that, for instance, input data in_1 may consists of 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 $T = \{T_i\}_{(i=1,N)}$, let $IN = \{in_l\}_{(l=1,L)}$ be the set of relevant input data sets. A task T_i's number of misses $miss_i(c_i, in_l)$ depends on task's allocated L2 size c_i and on the input data in_l. When the application processes the input data in_l, its number of misses, is denoted with M_l and it is given by the following:

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

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:

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

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 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:

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

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

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

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

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

If the stability is close to 100% the application behaves good for all its representative input data, so it is externally robust. If the difference between $\overline{M_l}$ and M_l 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 [4]. There are mainly two types of 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 [8] [9]. In these schemes the task that have either high priority [9] or large cache needs [8] dynamically "steals" cache ways from the other tasks. The purpose is to increase performance of high priority tasks [9] or to improve the overall hit rate [8].

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 used a CAKE multi-processor platform [3] with 4 Trimedia processor cores and 4 ways associative L2 cache. 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). We use the set-based L2 partitioning described in [6]. The experimental workload consists of two video multi-tasking applications: an H.264 decoder and a picture-in-picture-TV (PiPTV) decoder. We adapted the L2 cache sizes to the applications' requirements [6], while taking in consideration that typical L2 sizes for the CAKE platform are around 1-2 MBytes [3]. For the H.264 decoder the used L2 cache size is 1MB, and for the PiPTV decoder the used L2 cache size is 2MB. We executed these applications with standard definition input test sequences. We used the stimuli available at [7], which exhibit different degree of detail and movement. To have insight in the dynamic behavior of an input stream, we also investigate the stability variations among different number of frames of the same stimuli.

The data communication and synchronization among the tasks is done through FIFOs. The tasks are switched in two cases: (1) when they have no available input data or output buffer space or (2) when an interrupt occurs. 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 Section 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.

In the remainder of this section both application are briefly described and then the robustness assessment methods we introduced in Section 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 H.264

The H.264 decoder consists of several tasks [11]. 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 (Figure 3). The total number of tasks of this application is 15.

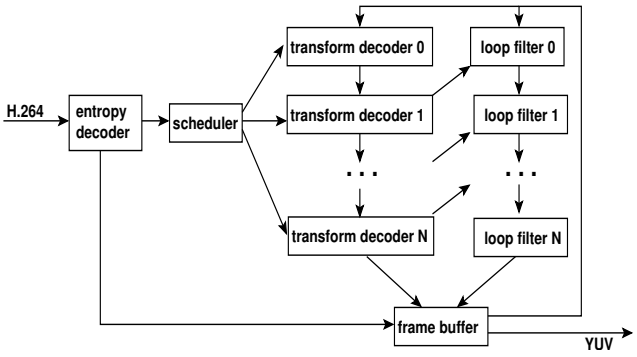

Figure 3: H.264 parallel application's tasks

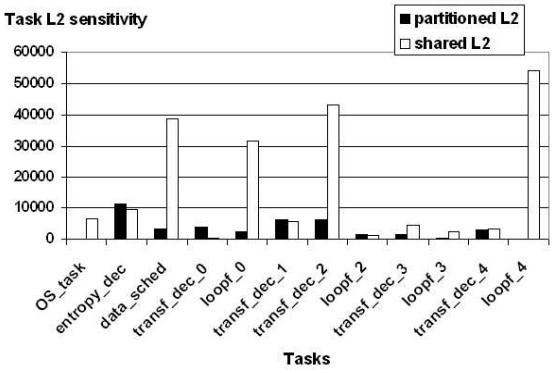

Figure 4: H.264 tasks L2 misses variation with task switching rate: shared vs. partitioned cache

To investigate the internal robustness we apply the technique in Section 3.1. The investigated average task switching range is from 41K times per second (corresponding to 4KB FIFOs) to 74K times per second (corresponding to 0.5KB FIFOs). For FIFOs larger than 4KB the average task switching rate does not decrease anymore because the value intrinsic to the application is reached. For FIFOs smaller than 0.5KB the application deadlocks, so the average task switching rate cannot be increased anymore. This task switching variation accounts for 30% difference in the number of L2 accesses.

The L2 sensitivity of tasks is compared for the partitioned and the shared cache case (Figure 4). In Figure 4 are depicted only the tasks with L2 misses variation larger than 0.5% of the H.264's overall misses. It can be observed that, in general, the shared L2 is more sensitive than the partitioned one. There are few tasks for which the sensitivity of the partitioned L2 cache is larger than the one of the shared cache. However, for all those tasks the sensitivity is smaller than 0.5%, so they can be considered irrelevant. Over all the application, the shared cache is 5 times more sensitive than the partitioned one. For a partitioned cache, over the investigated task switching range, the application's sensitivity as defined in Section 3.1 is at most 8%.

In the analysis of H.264 external robustness we found that the differences among the best partitioned ratio corresponding to different input data are relatively small. Across different input data the cache sizes for the transform decoders and loop filters vary with 32 cache sets (16KB of cache) which

input data	mobcal	parkrun	shields	stockholm
H.264 stability	96%	96%	100%	98%

Table 1: H.264 stability for different input data

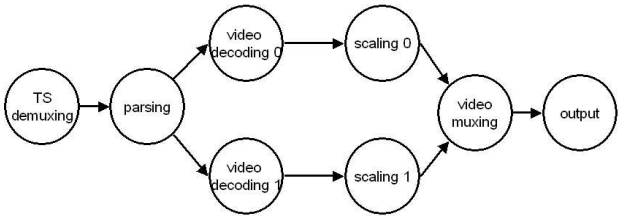

Figure 5: PiPTV parallel application's tasks

represents 1.5% of the entire cache. The cache size allocated to the entropy decoder task is always the same. Table 1 depicts the stability corresponding to each input data stream investigated. For some input data the partitioning ratio is non-optimal and this induces a performance degradation of maximum 4%. This corresponds to a stability of 96%, as defined in Section 3.2. Taking this facts into account, we can conclude that the static cache partitioning for the H.264 application is robust to input stimuli. A stability comparison between the shared and the partitioned cache is not possible because the stability, as defined in Section 3.2, is linked to the partitioned ratio, thus it cannot be computed for the shared cache scenario.

4.2 PiPTV

The picture-in-picture-TV (PiPTV) application decodes two different video streams and outputs raw pictures containing both video stream images, scaled with a given factor. This application consists of the following tasks (Figure 5): video demultiplexing of transport stream, two mpeg2 decoders (every one having multiple tasks [12]), two video scalers, video multiplexing the two images, and output. The PiPTV is described in YAPI and it is based on the work in [13]. The tasks connected in the graph depicted in Figure 5 are communicating data using FIFOs.

To investigate the internal robustness we vary the task switching rate. For every of the 4 Trimedia cores, the average task switching range is varied from 24K times per second (corresponding to 2KB FIFOs) to 41K times per second (corresponding to 0.4KB FIFOs). This task switching range accounts for 66% variation in the number of L2 accesses.

The L2 sensitivity of tasks is compared for the partitioned and the shared cache case. We depict in Figure 6 the L2 task sensitivity values for the tasks that experience the largest L2 variations. As it can be observed in the figure, the shared L2 is more sensitive than the partitioned one for most of the tasks. The L2 misses variation of tasks that are an exception from the previous observation are actually very small (in the range of 0.1% of the application's L2 misses). For the entire PiPTV application, the shared cache is 10 times more sensitive to L1 variations than the partitioned one. In the case of a partitioned cache, over the investigated task switching range, the application sensitivity, as defined in Section 3.1, is at most 5%.

For the PiPTV application the differences among the data dependent best partitioned ratio are at most 8 cache sets per

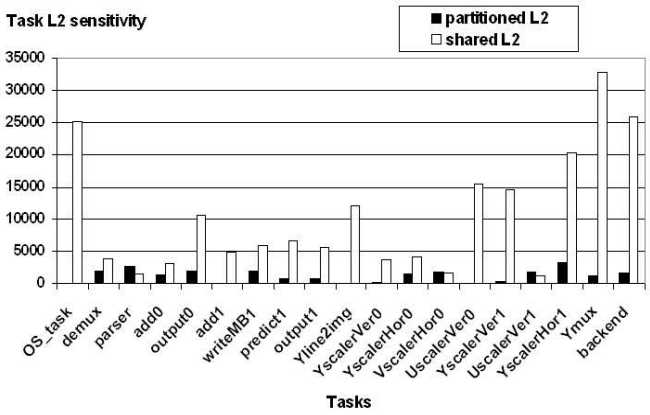

Figure 6: PiPTV tasks L2 misses variation with task switching rate: shared vs. partitioned cache

input data frames	10	15	30	60
PiPTV stability	92%	100%	93%	98%

Table 2: PiPTV stability for different input data

task. The total best cache ratio varies with 80 sets, which represents 7% of the total cache size. This partitioning ratio variations correspond to a maximum performance degradation of 9%. This corresponds to a stability of 91%, as defined in Section 3.2. We also exemplify the analysis of inside input stream dynamic behavior in Table 2. This table presents the stability figures corresponding to different number of frames from an input stimuli. In this case, we can observe that the minimum stability is 92%. Therefore, we can conclude that the static cache partitioning for the PiPTV application is robust to input stimuli. As already mentioned, the stability is defined in relation to the partitioning ratio, therefore the shared vs. partitioned cache comparison is not applicable for this metric.

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: (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 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 range 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 parallel applications: a picture-in-picture video decoder and a H.264

decoder and we used as simulation platform a CAKE multiprocessor instance [3]. In the internal robustness case, if the cache is partitioned, the H.264 sensitivity is 8% and the PiPTV sensitivity is 5%. Comparing the internal robustness of the shared and partitioned cache cases, we found that the shared cache is 5 times for the H.264 decoder, respectively 10 times for the PiPTV decoder, more sensitive than the partitioned one. This is an interesting fact on itself, because it suggests that the optimizations processes for L1 and L2 caches can be decoupled if the L2 is managed on a task centric manner.

The variations induced in the L2 behavior by various input data sets are at most 4% for the PiPTV application, respectively 9% for the H.264 decoder. This accounts for a stability of at least 96%, respectively 91%, therefore, for the investigated applications, we can conclude that the static cache partitioning is quite robust with respect to input stimuli variations.

6. REFERENCES

[1] A. Stevens, "Level 2 Cache for High-performance ARM Core-based SoC Systems", ARM white paper, 2004

[2] B.A. Nayfeh and K. Olukotun, "Exploring the Design Space for a Shared-Cache Multiprocessor", In Proceedings, ISCA, pages 166-175, 1994

[3] 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

[4] 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

[5] J.L. Hennesy and D.A. Patterson, "Computer Architecture: A Quantitative Approach", Morgan Kaufmann Publishers, 2003

[6] 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

[7] ftp://ftp.ldv.e-technik.tu-muenchen.de/pub/test_sequences/

[8] 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

[9] 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

[10] 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

[11] 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

[12] 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

[13] 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