Throughput optimization via cache partitioning for embedded multiprocessors

A.M. Molnos[(*)(**)] S.D. Cotofana[(*)] M.J.M. Heijligers[(**)] J.T.J. van Eijndhoven[(**)]

[(*)] Delft University of Technology
Mekelweg 4, 2426 CD Delft, The Netherlands
molnos@natlab.research.philips.com

[(**)] Philips Research Laboratories
High Tech Campus 5, 5656 AE
Eindhoven, The Netherlands

Abstract

In embedded multiprocessors cache partitioning is a known technique to eliminate inter-task cache conflicts, so to increase predictability. On such systems, the partitioning ratio is a parameter that should be tuned to optimize performance. In this paper we propose a Simulated Annealing (SA) based heuristic to determine the cache partitioning ratio that maximizes an application's throughput. In its core, the SA method iterates many times over many partitioning ratios, checking the resulted throughput. Hence the throughput of the system has to be estimated very fast, so we utilize a light simulation strategy. The light simulation derives the throughput from tasks' timings gathered off-line. This is possible because in an environment where tasks don't interfere with each other, their performance figures can be used in any possible combination. An application of industrial relevance (H.264 decoder) running on a parallel homogeneous platform is used to demonstrate the proposed method. For the H.264 application 9% throughput improvement is achieved when compared to the throughput obtained using methods of partitioning for the least number of misses. This is a significant improvement as it represents 45% from the theoretical throughput improvement achievable when assuming an infinite cache.

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 in an on-chip memory.

A possible organization of the on-chip memory which alleviate 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 [15], shared level two (L2) caches are used [12], [22]. 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 compositionality 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 [10]. In this approach, the compositionality is induced by allocating parts of the L2 cache, exclusively, to each individual task in the application. Hence, the cache partitioning ratio is an important parameter that can be tuned to optimize the application performance.

The existing methods for optimizing the partitioning ratio [9] [19] [11] minimize the overall number of misses. However, for media applications throughput is a more suitable optimization criterion. We consider applications consisting of a graph of communicating tasks. The throughput of such an application is bounded by the longest path in the task graph that has to be executed sequentially due to data dependencies (critical path). Minimizing the overall number of misses, does not necessarily minimize the critical path (improve the application throughput) and therefore the existing methods are less suitable for media task graphs.

In this paper we propose a method to determine the cache

partitioning ratio corresponding to the maximum throughput. This is a resource partitioning (allocation) problem, which is known to be NP hard, therefore we utilized a Simulated Annealing (*SA*) [7] strategy. During the *SA* process the throughput of the system has to be determined in order to decide if a partitioning ratio is a potential candidate for optimum. In our system we chose for flexibility and natural load balancing, therefore the scheduling policy is dynamic. In this case, the throughput cannot be analytically predicted. Hence, simulation is required in order to obtain the throughput value required by the annealing evaluation stage. An usual simulation of the multiprocessor platform is accurate, but too slow to be performed at every annealing step. Instead of a regular simulation, we use a fast, light simulation of the system. This means that only the synchronization is simulated to ensure the proper inter-task scheduling, whereas the rest of the instructions are only accounted for their execution time. This is possible because in a compositional environment tasks don't interfere with each other, therefore once their performance profiles are known, they can be used in any possible combination.

The application used to demonstrate our approach is H.264 video decoding, which is part of the newest video coding ITU-T standard [2]. The target architecture is a homogeneous multiprocessor platform [17], with an L2 cache which is partitioned among tasks. We investigate the proposed optimization method for different L2 cache sizes. The largest throughput improvement is experienced, for the smallest cache case (512KB). In this case, the H.264 throughput is improved with 6% comparing with the shared cache case and with 9% comparing with the cache partitioned for the least number of misses case. We note here that a maximum of 20% improvement (comparing with the shared cache case) is theoretically possible by assuming an infinite L2 cache. In the view of this observation, the proposed throughput optimization strategy delivers 45% of the possible throughput improvement, while keeping the same small cache size. In terms of miss rate, our method brings an absolute value increase of 4% when compared with the miss rate of the cache partitioned for the minimum number of misses.

The remainder of the paper is organized as follows. The related work is discussed in Section 2. Background information over the considered multi-processor platform and the cache partitioning method are introduced in Section 3. Section 4 describes the proposed throughput optimization method, and Section 5 presents practical experiments and results. Finally, the paper is concluded by Section 6.

2. Related work

Cache partitioning on itself is not new. In the literature multiple methods to partition the cache for decreasing the miss rate are described [16], [19], [11]. Few cache partitioning methods that improve power [4], allow prioritizing of tasks [20] or reconfiguration of cache [13] are also proposed.

In [19] the authors use an on-line associativity based partitioning scheme, estimate the miss characteristics of each process in hardware and dynamically partition the cache such that the overall number of misses is minimized. They target time sharing environments. The same authors present cache aware job scheduling in [18]. In [16] the problem of optimal allocation of cache between two competing processes that minimizes the overall miss rate is discussed. These approaches use associativity based partitioning (called column caching in their work) and they do not target compositionality, nor throughput increase for media applications. For performance reasons, typically the associativity of large caches is small, so column caching has too low granularity to be able to allocate exclusive cache parts to all tasks and shared data of the system such that compositionality is achieved.

In [11] and [6] a compositional data (respectively instructions) cache organization is proposed. A direct mapped cache can be partitioned and configured at compile time and controlled by specific cache instructions at run time, bringing considerable decrease in the number of misses. The main drawback of these approaches is that it is restricted to direct mapped caches.

Techniques to dynamically repartition the cache are presented in [13], [4], and [20]. In [13] a reconfigurable cache strategy is presented. Parts of the cache can be used for different processor activities and the authors evaluate instruction reuse, obtaining improvements in instructions per cycle. In [4] a possibility to disable a subset of the ways to save power is presented. Energy vs. performance trade-off is flexible and can be dynamically tailored. In [20] a prioritized cache for multi-tasking real-time system is presented. Using the prioritized cache the Worst Case Execution Time can be estimated more precisely and the cache miss rate decreases for the critical application. However, in these papers the authors do not target the compositionality of the system neither attempt to optimize the throughput of the system. Moreover, associativity based partitioning is used, with the already mentioned disadvantages.

Our work contributes to the state-of-the-art cache management domain in two aspects: (1) we propose a cache partitioning method that increases the application throughput in embedded chip multiprocessors. (2) based on the compositionality induced by our cache partitioning, we propose a fast throughput estimation method for multi-tasking applications with flexible, dynamic task scheduling policy.

3. Background

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

3.1 Target architecture

The envisaged multi-processor architecture consists of a homogeneous network of computing tiles on a chip [22]. 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. In this paper we use one tile of the multi-processor like the one depicted in Figure 1.

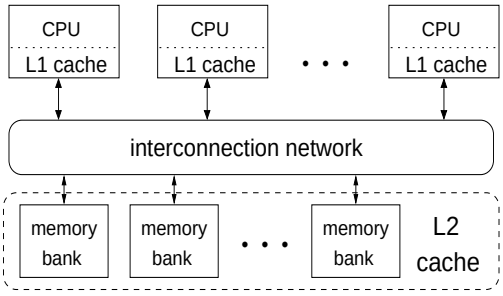

Figure 1. Multi-processor target architecture

The applications executed on this architecture consist of a graph of tasks that communicate through the memory hierarchy, thus through the shared L2. The inter-task synchronization is done by means of FIFOs. A task temporarily stops its execution in two cases: (1) when the task reads from an empty FIFO or (2) when the task writes in a full FIFO. Inter-task data exchange can be realized either through FIFOs, or using common memory regions. If several tasks share buffer regions, the synchronization at data access have to be taken care by the application programmer. The synchronization at common data is realized by communicating acknowledge-like tokens or data pointers through FIFOs. For example, a typical case of common regions are the reference frames of a video application. These frames are large (a state-of-the art high definition frame can be $\approx 2MBytes$) and transporting them multiple times among tasks is more expensive than just having a common copy of them. However, we use the term "common region" in a generic manner. For instance, a common buffer can be also the shared code of tasks that execute the same instructions on different parts of the data. For this shared code, no access synchronization is needed because the instructions are only read.

We choose for a task to processor mapping policy in which tasks may freely migrate from one processing unit to another, depending on these units availability. So overall, on the multiprocessor platform the tasks are executed in a pipelined fashion. The advantage of such a mapping policy is that it supports natural load balancing among processors. The disadvantage of this free scheduling and mapping policy is that, for the general case, the throughput formula cannot be analytically derived from the execution time of the tasks.

3.2 Cache partitioning

In the considered multi-tasking environment it is possible that two tasks T_i and T_j have 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 common buffers using the method introduced in [10].

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 common buffer gets an exclusive part of the cache sets. Second, inside the cache sets of a common buffer each task accessing it gets a number of ways.

In the rest of this paper we assume that an application A is composed out of N tasks, $T = \{T_i\}_{i=1,N}$ and M common region instances $R = \{R_j\}_{j=1,M}$. On the considered multi-processor platform, the L2 cache partitioning ratio CPR is the set of the cache sizes c_k allocated to every task T_i and common regions R_j: $CPR = \{c_1, c_2, ..., c_{N+M}\}$. Due to implementation efficiency reasons, the cache sizes have to be a power of two number of cache sets. When the actual cache size is relevant for the paper understanding, we designate the cache sizes with 2^n, otherwise we use just c_k.

On the targeted multiprocessor platform, we partition only the large level two cache because the most inter-task flushing occurs at this level. The L1 cache is small and belongs to a processor, so it can be considered private to the task that executes on that processor.

4. Throughput optimization

This section presents our method which optimizes the throughput via cache partitioning.

4.1. Throughput optimization problem

Typically, media applications have to process a certain amount of data before a time deadline. Therefore, for such an application, the throughput is defined as being the amount of data units processed in a time unit (for example, real time video decoding may require 25 frames per second). We denote with E_A the execution time needed by the application A to process one of its data units. Then, the throughput of the application (Th_A) is the inverse of the execution time needed to process a data unit: $Th_A = \frac{1}{E_A}$.

In this article we tackle the problem of finding the partitioning ratio CPR that gives the best throughput. This problem is similar with a capital partitioning problem, in which every production unit (tasks and common buffers) gets a number of resources (cache) such that the returned value (throughput) is maximum, under the constraints of a limited budget (total available cache). This is a known NP complete problem, therefore an heuristic should be deployed. Simulated annealing [7] is a well-known, powerful technique for combinatorial optimization problems, like for instance resource partitioning. To solve the throughput optimization problem we use it in the form presented in next section.

4.2. Simulated annealing

The Simulated Annealing (*SA*) optimization process used in this paper has the followings stages:

- *Initialization.* During this phase the temperature is set to a high value, Θ_0. The current partitioning ratio CPR_{curr} is initially set to a random value, CPR_0. The throughput of the application Th_{curr} corresponding to CPR_0 is determined using the light simulation method presented in detail in Section 4.3.

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

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

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

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

  This means that if the new throughput is larger than the current throughput, the current ratio is updated to the candidate ratio. Otherwise, if the new throughput is smaller than the current throughput, still some new candidates solutions are accepted in order to increase the chance of finding a global optimum and not "falling" in a local one. However, as Equation (1) suggests, if the temperature cools down, the chances of accepting a worse candidate are diminishing.

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

4.3. Light simulation

At each and every step of the *SA* optimization the throughput of the system has to be determined in order to decide if the current partitioning ratio is a potential optimum candidate. As we have chosen for flexibility and natural load balancing (therefore the scheduling policy is dynamic), the throughput cannot be analytically formulated. Therefore, simulation is required in order to obtain the throughput value needed for the SA evaluation stage.

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

light simulator is implemented using the CASSE tool chain [14], which is a System C [3] based tool.

In order to perform a light application simulation, one has to know the execution times and the FIFO read/write sequences of all tasks. For this, before the *SA* optimization, we gather tasks traces from regular simulation. A task's trace is a timed list of "execute" (e), "read from FIFO" (r), "write to FIFO" (w), and "access to common region" (a) actions. In the following paragraph we present the details regarding the e, r, and w actions. The a action is a special one and it is detailed in a separate paragraph.

An e action has assigned the time spend in execution. This time depends on the size of the cache part allocated to the task. During the light simulation an e action is behaviorally similar with a System C *"wait"* statement [3]. The *r/w* events have associated the *FIFO id* involved in the read (write) operation, the number of tokens consumed (produced) and the time spend to execute this operation. These informations are used to actually execute the FIFO reads and writes such that in the light simulation the same inter-task schedule is imposed as in the regular simulation. To gather the traces, each and every task T_i is accurately simulated with the list of T_i's possible cache sizes. The cache sizes have to be a power of two number of cache sets, therefore, the following are the possible cache sizes corresponding to task T_i: $2^0, 2^1, 2^2, ..., 2^{k(i)}$, where $k(i) \in N$ and $2^{k(i)}$ gives the maximum cache value for task T_i. This $2^{k(i)}$ is chosen such that if the tasks has $2^{k(i)+1}$ cache sets, no changes in its performance can be experienced. The trace of a task T_i contains the timing information of that task having a certain amount of cache, or in other words, the timing information of a (T_i, c_i) pair.

The access of common region buffer (the a action) is accounted separately in the execution time of a task because the access time depends on the cache allocated to the common buffer, not on the task itself. For every task t_i that accesses a common buffer R_j we collect the access time of a (T_i, R_j, c_j) tuple, where c_j is the cache allocated to buffer R_j.

As implied by the compositionality property, the tasks don't influence each other, therefore the timing of a (T_i, c_i) pair (or (T_i, R_j, c_j) tuple) can be used in every combination with the rest of the tasks. Hence, the throughput of a potential candidate $\{c_1, c_2, ..., c_{N+M}\}$ can be determined by a light simulation of the corresponding (T_i, c_i) and (T_i, R_j, c_j) traces.

5 Experimental results

For our experiments we used a CAKE multi-processor platform [22] with 3 Trimedia processor cores and 4 ways associative L2 cache. We use the L2 partitioning strategy described in [10]. The experimental workload consists of a multi-tasking video H.264 decoder [21]. We executed this application with standard definition input test sequences. We used the stimuli available at [1], which exhibit different degree of detail and movement. The results represent an average among the performance encountered for different stimuli.

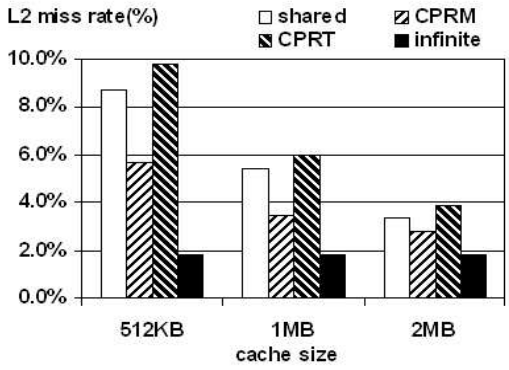

Figure 2. H.264 miss rate

The used parameter values of the SA optimization process are: initial temperature $\Theta_0 = 10^4$, the maximum cache excedent $\Delta C = 512KB$ the limit number of iteration without an optimum change $I = 100$. The temperature decreases at every step with a parameter $\alpha = 0.9$.

In the remainder of this section we first present the results of the throughput optimization method and then we evaluate and discuss the accuracy of the light-weighted simulation used in the *SA* optimization.

5.1 Throughput optimization

To validate our method, we compare the average L2 miss rate and time to process 25 frames for four cache configurations: (1) the cache fully shared, (2) the cache partitioned such that the number of misses is minimized (CPR_M), (3) the cache partitioned such that the throughput is maximized (CPR_T), and (4) the cache shared, but having an infinite size. In the H.264 case, the infinite cache size is approximated in practice with 4MB, because for cache sizes larger than that the miss rate (so the execution time) does not decrease anymore. Figures 2, respectively 3 depict the miss rates and completion time (for 25 frames) for the H.264 decoder in the four studied cache configurations, corresponding to different realistic cache sizes.

Looking at the CPR_M and the CPR_T cases, one can observe that the throughput is, as expected, larger for the CPR_T case at the expense of increased cache misses. The largest difference appears in the case of the smallest investigated cache (512KB), so we comment first on the results obtained with this L2 size case. The CPR_M cache configuration has an absolute L2 miss rate with 4% smaller

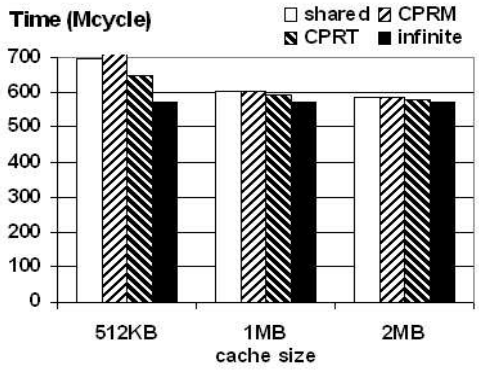

Figure 3. H.264 time to process 25 frames

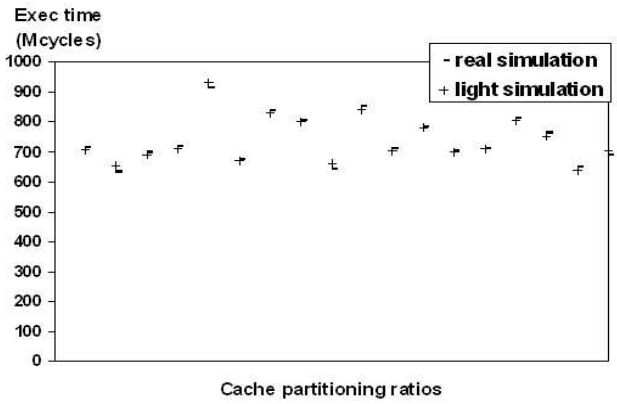

Figure 4. Light simulation accuracy

than the CPR_T configuration, but it is 9% slower when processing 25 frames. This throughput improvement of the CPR_T configuration corresponds to the completion of approximately two extra frames per second. When using an infinite L2 cache it can be observed that, the H.264 application has a 20% speedup when compared with 512KB, CPR_M cache configuration. These 20% represents the maximum speedup achievable by tuning the L2 cache. One can observe that the proposed throughput optimization strategy brings 45% of the possible throughput improvement, while keeping the same small cache size. When compared to a shared cache of 512KB the CPR_M cache configuration degrades the throughput with 3% (relative to the shared cache throughput) but the absolute miss rate decreases with 3%. The CPR_T cache configuration improves the throughput with 6% relative to the shared cache, but has an absolute miss rate with 2% higher. For the rest of the cache sizes (1MB and 2MB) the differences in performance among the four cases are not that large (under 2%).

The presented method is not restricted to the H.264 application. Every parallel application running on a multiprocessor system can benefit from the proposed optimization.

Two phenomenon justify the difference in misses' number between a shared and a partitioned cache. If the cache is partitioned, the inter-task cache flushing is eliminated (which means less misses) but every task can use less cache space than in the shared case (which means more misses). The variation of the execution time with the number of misses is not linear because by minimizing the overall number of misses the sum of tasks execution times is minimized. However, because the tasks are executed in parallel the critical path in the application gives the overall completion time and it is not the sum of tasks execution times.

5.2 Light simulation

As introduced in the Section 4, the *SA* performs a light simulation of the system to evaluate the throughput corresponding to a given partitioning ratio. In the following we present an accuracy investigation of this light simulation method. Figure 4 depicts the average completion time for H.264 decoding of 25 frames in two cases: regular simulation and light simulation. The comparison between this two cases is presented for multiple cache partitioning ratios. The maximum difference between the completion time reported by the regular simulation when compared with the light simulation is 3%. Worth to mention is that the light simulation is at least 30 times faster than the regular simulation. The 3% difference is actually caused by the fact that the system is not 100% compositional. Some tasks' timings are slightly different from a configuration to the other because the L1 cache is not partitioned. The L1 is considered private to each and every task during its execution. However, there are variations in the cache access pattern due to L1 behavior. For more insight in the compositionality and robustness of our cache partitioning scheme we refer the reader to [8].

We would like to stress out the fact that the light simulation method as described in the Section 4.3 is applicable due to the compositionality induced by cache partitioning. In a compositional environment the timed traces of tasks can be gathered once and used in any possible combination, because tasks don't interfere with each other.

6 Conclusions

In this paper we proposed a cache partitioning method to maximize the throughput of a multi-tasking application mapped on an embedded multiprocessor. Our method assumes an on-chip multiprocessor platform with a large shared L2 cache and improves throughput by tuning the L2 cache sizes allocated to each and every task. Because resource allocation problems are NP complete, our method is based on a simulated annealing strategy. At every step of

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

To demonstrate our method we utilized an H.264 multitasking decoder. For the H.264 application a 9% throughput improvement is achieved when compared to existing methods of cache partitioning for the least number of misses. When compared with the shared cache case 6% throughput improvent is achieved under the circumstances that a maximum of 20% is actually possible by having an infinite L2 cache. This implies that the proposed throughput optimization strategy brings 45% of the possible throughput improvement, while keeping the same small cache size. The miss rate when using our method increases with an absolute value of 4% when compared with the cache partitioned for the least number of misses case.

References

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

[2] International Telecommunication Union, http://www.itu.int.

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

[4] D. H. Albonesi. Selective cache ways: On-demand cache resource allocation. *International Symposium on Microarchitecture*, 1999.

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

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

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

[8] A. Molnos, S. Cotofana, M. Heijligers, and J. van Eijndhoven. Static cache partitioning robustness analysis for embedded on-chip multi-processors. *In Proceeding of the ACM International Conference on Computing Frontiers*, To appear 2006.

[9] A. Molnos, M. Heijligers, S. Cotofana, and J. van Eijndhoven. Compositional memory systems for multimedia communicating tasks. *Proceedings, DATE*, pages 932–937, 2005.

[10] A. Molnos, M. Heijligers, S. Cotofana, and J. van Eijndhoven. Compositional, efficient caches for a chip multiprocessor. *Proceedings, Design, Automation and Test in Europe*, To appear in 2006.

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

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

[13] P. Ranganathan, S. Adve, and N. P. Jouppi. Reconfigurable caches and their application to media processing. *Proceedings, 27th Annual International Symposium on Computer Architecture*, 2000.

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

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

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

[17] P. Stravers and J. Hoogerbrugge. Homogeneous multiprocessing and the future of silicon design paradigms. *Proceedings, International Symposium on VLSI Technology, Systems, and Applications (VLSI-TSA)*, April 2001.

[18] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. *HPCA*, pages 117–, 2002.

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

[20] Y. Tan and V. J. Mooney. A prioritized cache for multitasking real-time systems. *Proceedings of the 11th Workshop on Synthesis And System Integration of Mixed Information technologies*, pages 168–175, 2003.

[21] E. van der Tol, E. Jaspers, and R. Gelderblom. Mapping of h.264 decoding on a multiprocessor architecture. *Proceedings, SPIE Conference on Image and Video Communications and Processing*, 2003.

[22] J. T. van Eijndhoven, J. Hoogerbrugge, M. 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.