Inter-task cache sharing for compositional embedded multiprocessors

A.M. Molnos[(*)(**)] M.J.M. Heijligers[(**)] S.D. Cotofana[(*)] J.T.J. van Eijndhoven[(**)]
[(*)] Delft University of Technology Mekelweg 4, Delft, The Netherlands
[(**)] Philips Research Laboratories HTC 5, 5656 AE Eindhoven, The Netherlands
email: molnos@natlab.research.philips.com

Abstract— **In current multi-media systems a major part of the application consists of multiple software tasks executed on a set of processors. Composing the system performance out of the tasks performance is possible only if these tasks interfere with each other in a predictable way. However, when the used memory hierarchy consists of shared caches, tasks flush each others data out of the cache in a unpredictable manner. This paper proposes a novel cache partitioning technique that ensures performance compostionality combined with cache efficiency. We perform two partitioning types. First, each task and each inter-task common data gets a exclusive part of the cache sets. Second, inside the cache sets of common data each task accessing it gets a number of ways. We confirm the proposed method on a homogeneous multiprocessor using two applications: H.264 decoding and picture-in-picture-TV. Our experiments indicate that the difference between the sum of misses of individual tasks in isolation and the number of misses of the complete application is at maximum 3%, so we can conclude that compositionality is achieved. Additionally, when compared to the shared cache scenario the execution time is improved up to 16% and the miss rate is reduced with up to 9%.**

I. Introduction

The predictability is the main required characteristic for media applications for which guaranteeing the completions of tasks before their deadlines is of crucial importance. The low power and low cost demands of embedded domain make the use of general purpose architectures with clock frequencies in the order of several GHz inappropriate. Instead, in the embedded domain Chip Multi-Processor (CMP) architectures are preferred.

On a CMP platform the application tasks interfere with each other when they share memory and other hardware components. In order to be able to reason about the system performance, the performance of each individual task must be preserved if the tasks are executed concurrently in arbitrary combinations or if additional tasks are added. A system satisfying this property is addressed as being compositional, and we believe that multitasking systems cannot be predictable without being compositional.

Many state-of-the-art media applications process large data that reside off-chip. The availability of these data at the right moments in time is critical for the application performance. A possible solution for on-chip data availability is to use shared cache memories [7]. However, when using shared cache in conjunction with a CMP architecture and multi-tasking applications, the following case can occur: when a task T_i's data is loaded into the cache it may flush task T_j's data, eventually causing a future T_j miss. Therefore, shared caches ruin the compositionality of a CMP system.

Exclusive cache partitioning is an existing method to deal with the inter-task cache interference and to acquire systems compositionality [15]. However, when, for example, multiple tasks execute the same function on different data streams, exclusive cache partitioning leads to compositionality, but also to code multiplication in cache, so to poor cache usage. For example, when the instructions of several H.264 tasks are multiplied in cache, our experiments indicate a 50% increase of the number of misses, resulting in 10% execution time penalty compared with a conventional shared cache. Providing a shared cache part for the common instructions is not a viable alternative as it leads to tasks trashing each others data in that shared cache part, so to non-compositionality. As no principial difference between the two type of sharing exist, for simplicity we use in the remainder of this paper the term "common regions" for both inter-task shared data and instructions.

In this paper we extend the work in [15] by tackling the problem of sharing the common region cache while preserving the compositionality of the system. First, to ensure compositionality, we separate tasks' private and common data in cache by allocating cache sets for them. Subsequently, we eliminate the inter-task conflicts in the shared cache sets of common data, by allocating ways of those sets to tasks. So, overall, we utilize mixed (set and associativity based) cache partitioning.

We confirm the proposed method on a multiprocessor using two multi-tasking applications: H.264 decoding and picture-in-picture-TV. Our experiments indicate that for both examples, the difference between the sum of misses of individual tasks in isolation and the number of misses of the complete application is at maximum 3%, so we can conclude that compositionality is achieved. Additionally, for typical cache sizes, our method has positive impact in the overall performance. When compared to case of fully shared cache, the performance improve as follows: (1) the H.264 decoding exhibits 3% less L2 misses corresponding to a 5% execution time improvement and (2) the PiPTV applications experience 66% less L2 misses corresponding to a 16% execution time improvement.

The outline of this paper is as follows: in Section II the state of the art in the domain of cache partitioning is presented, in Section III the targeted system together with the proposed cache partitioning method are presented, in Section IV the implementation of the cache partitioning is presented, the practical experiments are presented in Section V and Section VI concludes the paper.

II. Related work

Cache partitioning on itself is not new. In the literature different (set or associativity based) cache management methods were proposed.

In [8] the authors use an on-line associativity based partitioning algorithm achieving interesting performance improvement. They estimate the miss characteristics of each process and partition the cache dynamically in order to minimize the number of misses. However, this approach cannot enable the performance compositionality mainly due to the fact that the associativity based partitioning has a too low granularity to be able to allocate exclusive cache parts to all tasks and common data of the system such that compositionality can be achieved.

The authors of [5] and [9] propose a compositional data (respectively instructions) cache organization. A direct mapped cache can be partitioned and configured at compile time and controlled by specific cache instructions at run time, considerably outperforming a conventional cache. For our purposes, the main drawbacks of this approach are that it is restricted to direct mapped caches and it is unclear if inter-task sharing of data (image frames of a video application for example) can be made compositional.

In [4] the cache is partitioned among tasks at compile and link time. In [2] a method to divide a cache into partitions for each real-time task and a larger partition called the shared pool for the non-real-time tasks is described. In both approaches the authors do not take into account tasks' common region, so they are not applicable for our environment.

Liedtke et al. propose in [3] a cache partitioning method controlled by the operating system. The major drawbacks of this method are the limitation to physically indexed caches and the basic partitioning unit assignable to a task of one memory page.

The present work differs from existing approaches in the sense that we focus on achieving performance compositionality for application executed on multiprocessor platforms. Compositionality is a desired property because it increases the system predictability and it decreases the engineering complexity. Efficient cache usage is a subsequent purpose and should not disturb the compositionality.

III. Sharing data and instructions with enabling compositionality

This section presents the proposed cache management technique for achieving performance compositionality and sharing the cache for tasks common data and instructions.

A. Target architecture

The envisaged architecture is the CAKE platform [10]. This platform consists of a homogeneous network of computing tiles (like the one in Figure 1) on a chip. Each tile contains CPUs (Trimedia and/or MIPS cores), a router (for out of tile communication), and memory banks. The processors are connected to memory by a fast, high-bandwidth interconnection network. The on-tile memory is actually used as a unified L2 cache, shared between processors, facilitating a fast access to the main memory which is outside the chip. In this paper we use one tile of the multiprocessor. On such a tile, the CAKE platform implements a cache coherence protocol among the different L1's and L2.

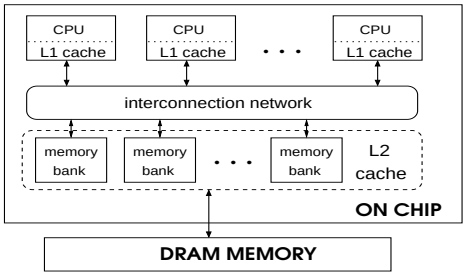

Fig. 1. Multiprocessor target architecture

The applications executed on this architecture consist of sets of tasks that communicate through the memory hierarchy, thus through the shared cache. Three types of parallelism are possible in multimedia applications: functional parallelism (where tasks perform different operations on the same data input), data parallelism (where task performs the same operation on different parts of the input data), and a mix of the two previous ones. In the case of data parallelism multiple tasks execute the same instructions on different parts of the input data. Moreover, independently of the parallelism type, multimedia task usually share variables (for example reference frames for video codecs). Thus, in media applications tasks share code an data, denoted in this paper with "common regions".

Our work targets conventional, set associative caches [6]. Such a cache is a rectangular array of memory elements arranged in "sets" (rows) and "ways" (columns). The set where a data instance can be placed is uniquely identified by a part of the data address. Inside that set, all the ways are searched to determine if the data is present in cache. If a data instance is not found in cache, it should be loaded. Loading of new data in a cache set implies that, if the set is full, data already existing there is swapped out of the cache. In a multi-tasking environment is possible that two tasks T_i and T_j have data mapped in the same cache set. Therefore, when T_i's data is loaded into the cache it may flush task 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.

B. The proposed cache partitioning

A common method to achieve performance compositionality is by allocating to each task its own exclusive cache part. But when tasks have common regions, a copy of such a region will reside in the cache part of every task using it, causing coherence problems and having a negative impact on cache utilization. Another option is to provide a shared

cache partition for every common region, but then its compositionality cannot be achieved due to the inter-task cache trashing in that shared cache part.

Our method first ensures that the instances of private tasks data and common regions don't trash each other in cache. For that we restrict the cache sets used by every task and every common region. We address this technique as "set based cache partitioning". Subsequently, we create the premises such that tasks don't trash each other data in the cache sets of the common regions. For the cache sets sharing problem we present two possible solutions:

- The cache allocated to the common data is as large as the data instance itself. In this case no misses occur, hence no unpredictable trashing is present.

- Inside the cache sets of a common region tasks use the data if it is already there (sharing) but on a miss they are not allowed to flush other tasks ways (don't interfere).

The first solution depends on the application and on the available cache, so it is not always applicable. For instance, for the state of the art video definition reference frame buffers typically do not fit in the cache. The second solution is more general and can be applied regardless of the relation between the sizes of available cache and the common data. This general solution can be easily implemented by allocating to the tasks a number of ways in the sets a common region. Because the number of ways in the cache is denoted by "associativity" [6], we denote this partitioning type as "associativity based".

In conclusion, for achieving performance compositionality we use mixed cache partitioning like depicted in Figure 2. The dark gray cache part is allocated to task T_0 and the light gray cache part is allocated to task T_1. In the shared T_0 and T_1 cache region tasks can can query all the four ways of the corresponding cache set for a hit. However, if for example a T_1 access misses in cache, the replacement takes place only in T_1's two ways.

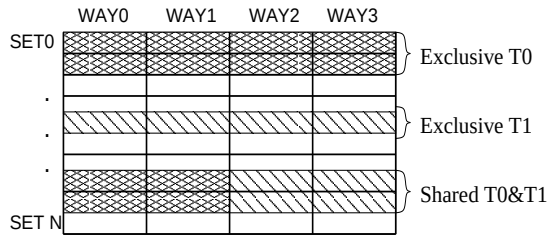

Fig. 2. Mixed cache partitioning

When using associativity based partitioning the tasks that access the common region should have each at least one way of the shared cache sets, so cache associativity should be greater or equal with the number of tasks sharing the common region. We note however that the maximum number of tasks that share a common region is typically smaller than the number of tasks forming an application. Future research will investigate the options to overcome this restriction.

We apply the cache partitioning on the L2 shared cache, because it is the most affected by the inter-task run-time conflicts and we make two assumptions. Since the levels of cache private to each processor are usually small and task switching rate in multimedia application is typically low enough, we assume that L1 cache can be considered private to each task. The second assumption is that our work is part of a large framework used to ensure compositionality, where every shared resource (busses, networks, memory banks, memory ports etc.) should be also managed for performance compositionality. Our work targets the cache strategy and ensure that the number of misses of every task is not disturbed by other tasks. Subsequent mechanisms should take care that a miss takes a limited number of cycles, independent on the concurrency or shared resource contention. In this paper we assume that these shared resources suffice such that they don't interfere in the compositionality of the system.

We chose the set based and not the associativity based partitioning to isolate different tasks and common regions footprints in the cache. The motivation for this choice is that typically in a cache there are more sets than ways, therefore set based partitioning allows every task to have its own exclusive part. In a large L2 cache, the state-of-the-art number of ways is around 16. If we would have used associativity partitioning for task footprint isolation and the application would have more than 16 tasks, some cache ways should have been shared among tasks, leading to the already presented inter-task flushing problem. Therefore, in general associativity partitioning cannot enable compositionaly except if the number of ways is smaller or equal with the number of task in the considered application. Moreover, the low granularity of the associativity partitioning limits the options of improving performance by tuning the partitioning ratio to the tasks requirements.

In the following we present the implementation issues first for set based partitioning and then for associativity based partitioning.

IV. Cache partitioning implementation

In the organization of conventional, set associative cache the address splits in three parts: "tag", "index" and "offset" [6]. The index directly addresses a cache set (row). Every set has a number of M ways (column). The tag part of the address is compared against all the tag parts stored in a set to determine if there is a hit in one of the set's ways. In the following subsections we present in detail the two types of partitioning that we apply. Given that we provide the mechanisms to support both set and associativity cache partitioning and the fact that their combination does not require additional steps, mixed partition is also supported.

A. Set based cache partitioning

In set based partitioning every task and every common region get a number of sets from the cache, as depicted is Figure 3 A. The set based cache partitioning requires translating the index part of the address such that it access another cache set than it originally did. The index translation mechanism is controlled by the task *id* (for a private tasks

data access), or by the common region *id* (for a common region access), as in Figure 4. To avoid expensive modulo operations, the partition sizes are limited to power of two number or sets. A table provides the $MASK$ and $BASE$ values for every task and common region. To clarify the mechanism, let us assume that an access to data A has the index idx_A if the cache would have been conventional. We denote by 2^k the size of partition for A and by 2^C the size of the total cache (both size values are considered in number of sets). The $MASK_A$ actually selects 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). The $BASE_A$ fills the rest of the $C-k$ index bits such that different tasks accesses are routed in disjoint parts of cache. After index translation, two addresses that didn't have the same old index might end up having the same new index. Therefore, the old tag and old index bits form the new tag used for correct cache lookup. Hence, every tag has 10-12 extra bits (depending on cache size), representing less that 1% of the total L2 area, so the penalty implied is negligible. The execution of the coherence protocol takes few cycles; therefore, in parallel with it, the index translation for L2 accesses can be performed. This parallel execution results in no additional delay penalty involved for the extra index translation.

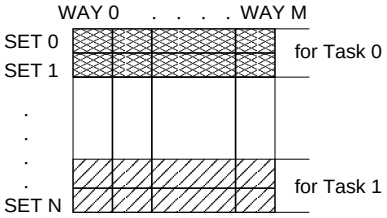

Fig. 3. Set based cache partitioning

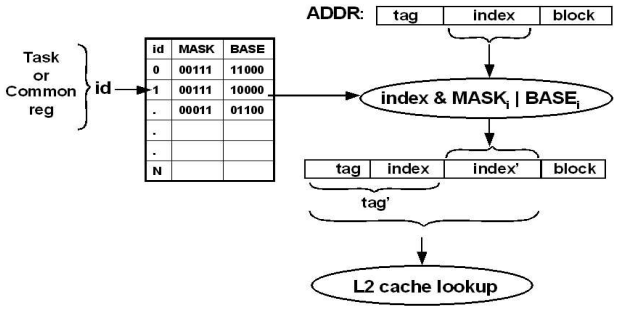

Fig. 4. Set based cache partitioning - implementation

In order to perform a correct exclusive set based partitioning each memory access should be labeled with its corresponding *task id* or *comm_reg id*. The *task id* for every processor is stored in a register and updated at every task switch, therefore it can be used directly. Common regions consist of data or code. In the following we present the options to obtain a common region *id* first for data and then for code.

There are several ways to obtain an *id* for the common task data. A *comm_reg id* register could be used, so the compiler should keep that register up to date. Alternatively, a part of the address could be used to encode the *comm_reg id*. This approach requires a cache aware memory allocator, reduces the usable address space (fragmentation), and also requires adapting the compiler for handling shared static data structures. Nevertheless, for dynamic memory allocation the partitioning can be implemented relatively straightforward by providing a dedicated malloc for shared buffers. A third approach is to keep a table with intervals of shared memory and for every access the cache can lookup if the address has an associated *comm_reg id*. This third approach is more expensive in terms of area and power. For our experiments we choose the third alternative because we are mainly interested in the system level aspects (e.g., inducing the compositionality, implication in miss rate). The third approach is more generic than the others because any address range can be placed in any place in the cache. This easily allows for other experiments, like for example separating tasks' instructions and static variables in the cache or sharing some cache partitions.

Using the same method as for shared data we can obtain a *comm_reg id* for the common code. However, this approach requires extra analysis to determine the address ranges of the common regions of code. Another option is to distinguish between code and data accesses, by labeling the L2 accesses coming from the L1 instruction cache as code. At compile time it is known which tasks are instantiated multiple time, therefore the entries of the index translation tables can be set such that the shared code accesses are routed into the same cache partition.

B. Associativity based cache partitioning

In associativity based partitioning a task gets a number of ways from a part of the cache sets, as depicted is Figure 5. This partitioning is called column caching [11]. Allowing every task to search all the cache ways for a hit (but in case of a miss to replace data only task's own ways) easily ensures sharing of common task regions.

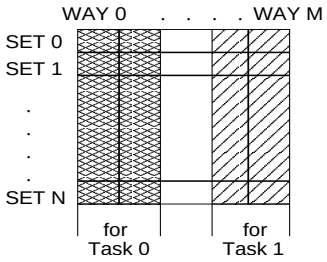

Fig. 5. Associativity based cache partitioning

The associativity based partitioning is implemented by changing the cache replacement policy in case of a miss. Depending on the *task id* only a restricted number of ways of one set are used for victimizing old data and bringing in missed data. The associativity based partitioning requires small additional logic and the penalty can be neglected.

In the existing light-weighted operating system respon-

sible with task scheduling we added primitives for loading and modifying the necessary tables and registering address ranges of common regions. During application initialization the designer can load the translation tables such that a desired cache partitioning is imposed.

V. Experimental results

For our experiments we used a CAKE multiprocessor platform [10] with 4 Trimedia processor cores and 4 ways associative L2 shared cache. The experimental workload consists of two multi-tasking applications: a H.264 decoder and a Picture-in-Picture-TV (PiPTV) decoder. Both applications exhibit mixed data and functional parallelism and are separately simulated on the CAKE platform. Nevertheless, our technique is not restricted to these applications. For instance, every data parallel multimedia application can benefit from instruction cache sharing in a compositional manner.

The H.264 decoder consists of several tasks [13]. First an entropy decoder task processes the input stream and passes the data via a scheduler to a set of transform decoders and loop filters tasks doing inverse quantization, transformation, prediction respectively deblocking on different parts of the image. The transform decoders and loop filters are data parallelized. They share the instructions and the reference frames.

The PiPTV consists of multiple tasks: two mpeg2 decoders, two video scalers, video multiplexing and demultiplexing. The application is described in YAPI and it is based on the work in [12].

Our experiments investigate two issues: (1) compositionality and (2) cache partitioning implications in system performance. The used partitioning ratio is chosen such that the overall application number of misses is minimized. The process of finding this optimized ratio has first an information gathering phase during which every task is individually simulated having different amounts of cache. Then the optimized partitioning ratio is computed by minimizing the sum of all task misses, under the constraint that all allocated cache is not larger than the available cache.

We study the compositionality using the variation between the sum of misses of individual tasks in isolation and the number of misses of the complete application. The misses of every individual tasks in isolation are obtained during the gathering phase. The number of misses of the complete application is obtained by simulating all the tasks together, using the optimized partitioning ratio. For both examples the variation are smaller than 3%, so we can conclude that compositionality is achieved within reasonable bounds. The 3% difference is due to the neglected effects like L1 presence, task switching and migration.

The performance implications of mixed partitioning are studied by comparing the L2 number of misses and execution time for two cache configurations: (1) the cache fully shared, and (2) the cache partitioned as proposed in this paper, with the partitioning ratio optimized for overall least number of misses. We execute the applications with standard definition test sequences having different degree of detail and movement [14]. In Figures 6, respectively 7, the average miss rate and completion time for the two studied cache configurations are presented for the PiPTV and respectively H.264 applications.

For both application one can observe that, for a small L2 size mixed partitioning degrades the performance of the cache with up to 7%. This results in execution time increase 20% for the PiPTV application and 25% for the H.264 decoder. For the rest of the considered L2 sizes, the partitioned cache outperforms or it is at least as good as the shared cache. In these cases, the average miss rate reduction of partitioned cache over the shared cache is at 6% for the PiPTV application and 3% for the H.264 decoder.

The typical L2 sizes for the CAKE platform are around 2 MBytes [10]. For this size the reductions in miss rate are as follows: 3% for H.264 and 9% for PiPTV. The miss rate reductions results in execution time improvement of 5% for the H.264, respectively 16% for the PiPTV.

Two phenomenons determine the number of misses difference between a shared and a partitioned cache. If the cache is partitioned, the inter-task cache flushing is eliminated (which means less misses) but every task can use less cache space than in the shared case (which means more misses). In our examples one can observe that for a small L2 size the second effect is dominant, whereas for larger L2's eliminating inter-task flushing leads to performance improvement. The variation of 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 overall completion time is given by the critical path in the application and it is not the sum of tasks execution times.

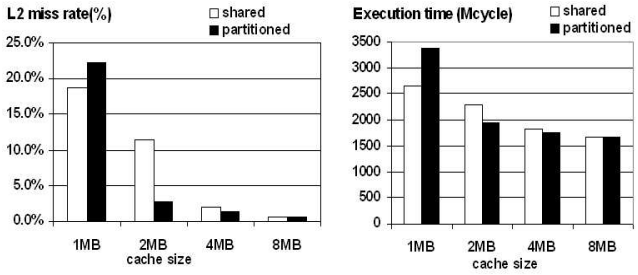

Fig. 6. PiPTV: shared vs. partitioned cache

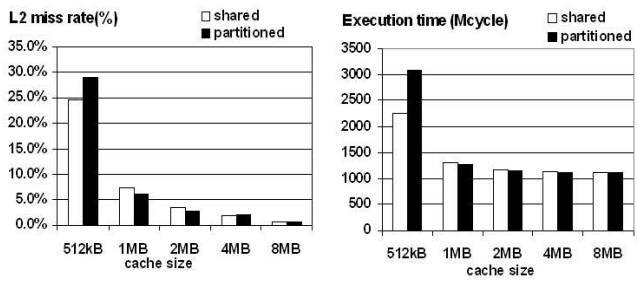

Fig. 7. H.264: shared vs. partitioned cache

VI. Conclusions

This paper proposed a method that contributes to the use of a multiprocessor with shared caches in real-time systems. We developed a set and associativity based cache partitioning technique that ensure performance compostionality within reasonable bounds and allows cache sharing for common tasks data and/or instructions. Apart from allowing the designer to predict the overall performance out of the performance the parts, compositionality enables also reuse and easy integration of tasks into systems, which decreases engineering efforts, therefore shortens the time to market.

Our method removed the inter-task cache interference by using two cache partitioning types. First, each task and each inter-task common data had allocated a exclusive part of the cache sets. Second, inside the cache sets of common data each task accessing it had allocated a number of ways. The proposed method was applied to the shared L2 cache of a CAKE multiprocessor. Two multi-tasking applications were used for the practical experiments: H.264 decoding and picture-in-picture-TV. Our experiments indicate that, for both applications, using our partitioning scheme the sum of misses of the individual tasks executed separately and the number of misses of all tasks executed concurrently differs at most by 3%, so we can conclude that compositionality was achieved within reasonable bounds. Additionally, for typical L2 sizes, the partitioned cache outperformed the fully shared cache leading up to 16% execution time reduction. Future work includes dynamic repartitioning strategies.

References

[1] 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.

[2] D.B. Kirk. SMART (Strategic Memory Allocation for Real-Time) Cache Design. In *IEEE symposium on Real Time Systems*, 1989.

[3] J. Liedtke, H. Härtig, M. Hohmuth. OS-Controlled Cache PRedictability for Real-Time Systems. In *3rd IEEE Real-Time Technology and Applications Symposium*, 1997.

[4] F. Mueller. Compiler Support for Software-Based Cache Partitioning. In *ACM SIGPLAN Notice*, 1995.

[5] H. Muller, D. Page, J. Irwin, D. May. Caches with Compositional Performance. In *Proceedings, Embedded Processor Design Challenges*, 2002.

[6] J.L. Hennesy, D.A. Patterson. *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, 2003.

[7] B.A. Nayfeh, K. Olukotun, "Exploring the Design Space for a Shared-Cache Multiprocessor", In *ISCA*, 1994

[8] E.G. Suh, L. Rudolph, S. Devadas, "Dynamic Partitioning of Shared Cache Memory", In *The Journal of Supercomputing*, 2004

[9] J. Irwin, D. May, H. Muller, D. Page, "Predictable Instruction Caching for Media Processors" In *13th International Conference on Application-specific Systems, Architectures and Processors (ASAP)* 2002

[10] J.T.J. van Eijndhoven, J. Hoogerbrugge, M.N. Jayram, P. Stravers, A. Terechko, "Cache-Coherent Heterogeneous Multiprocessing as Basis for Streaming Applications", In *In Dynamic and robust streaming between connected CE-devices*, to appear in 2005.

[11] D. T. Chiou "Extending the Reach of Microprocessors: Column and Curious Caching", *PhD thesis Department of EECS, MIT, Cambridge, MA*, 1999.

[12] E.A. de Kock, and all "YAPI: application modeling for signal processing systems", In *Proceedings, 37th conference on Design Automation* ,2000.

[13] E.B. van der Tol, E.G. Jaspers, R.H. Gelderblom. "Mapping of H.264 decoding on a multiprocessor architecture, In *Image and Video Communications and Processing*, 2003

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

[15] A.M. Molnos, M.J.M. Heiligers, J.T.J. van Eijndhoven, S.D. Cotofana "Compositional memory systems for multimedia communicating tasks", in Proceedings, DATE, 2005