# Compositional, efficient caches for a chip multi-processor

A.M. Molnos$^{(*)(**)}$ M.J.M. Heijligers$^{(**)}$ S.D. Cotofana$^{(*)}$ J.T.J. van Eijndhoven$^{(**)}$

$^{(*)}$ Delft University of Technology
Mekelweg 4, Delft, The Netherlands
molnos@natlab.research.philips.com

$^{(**)}$ Philips Research Laboratories
HTC 5, 5656 AE
Eindhoven, The Netherlands

## Abstract

*In current multi-media systems major parts of the functionality consist of software tasks executed on a set of concurrently operating processors. Those tasks interfere with each other when they share memory and other hardware components. For instance when the tasks share caches and no precautions are taken they potentially flush each other's data at random. In this case the control over the system performance is lost. However, in media processing the performance must be under tight control. In particular 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.*

*This paper proposes a novel cache partitioning technique that enhances compostionality. We assume a 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 data gets an exclusive part of the cache sets. Second, inside the cache sets of common data each task accessing it gets a number of ways. We apply the proposed method on a homogeneous multiprocessor using two applications: H.264 decoding and picture-in-picture-TV. Our experiments indicate that, for both applications, under 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 4%. We conclude that compositionality is achieved within reasonable bounds. Additionally, our technique appears to improve the efficiency of the cache operation.*

## 1. Introduction

In order to guarantee sufficient performance, the predictability is the main required characteristic for state-of-the-art media applications. 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. Many media applications process large data residing off-chip. The availability of these data at the right moments in time is critical for the application performance. A possible approach to cope with the problem of on-chip data availability is to use shared cache memories [11]. However, when used in conjunction with a CMP architecture and multi-tasking applications, shared caches make the miss rate prediction (thus performance control) difficult. For instance, when 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. This kind of unpredictability constitutes a major problem for real-time applications for which the completions of tasks before their deadlines is of crucial importance. In particular, 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 having *compositional* performance. Compositionality enables also reuse and easy integration of tasks into systems, which shortens the time to market, another important aspect for the embedded domain.

Cache partitioning among tasks is the most used approach to mitigate the inter-tasks interference in cache [9], [10], [12], [6]. The existing cache management schemes use one of the two types of cache partitioning, as follows: (1) set (row) based partitioning (exclusive cache sets are assigned to tasks) or (2) associativity (column) based partitioning (ways of every cache set are assigned to tasks). However, these existing methods cannot be straightforwardly extended for the case tasks share data and/or instructions.

In this paper we propose a novel cache partitioning technique that enhance performance compositionality and allows cache sharing for common tasks data and/or instructions. As no principial difference between the two types 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. Our method uses both set and associativity types of cache partitioning. First, we ensure that no task access may flush a common cache region or other task. This isolation is achieved by exclusively assigning a number of cache sets to every task and to common regions, via a set based partitioning process. Second, we propose a strategy to guaranty those tasks don't trash each other inside the cache sets allocated to a common region. Each task that shares a common region has assigned a number of ways in the sets allocated for that common region. This second strategy is realized via associativity based 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 4%, so we can conclude that compositionality is achieved. Additionally, for typical cache sizes, our method has positive impact in the overall performance.

The remainder of the paper is organized as follows. The state of the art in the domain of cache partitioning is presented in Section 2. The proposed cache management is introduces in Section 3 and issues related to its implementation are described in Section 4. Section 5 presents experimental results and Section 6 concludes the paper.

## 2. Related work

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

In [13] 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 [10] and [5] 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 [9] the cache is partitioned among tasks at compile and link time. In [6] 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 [7] 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.

In previous work [8] we tackled only the case of tasks that do not have common instructions.

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.

## 3. Sharing data and instructions with enabling compositionality

This section presents the proposed cache management technique for achieving performance compositionality and sharing the cache for common data and instructions among tasks. The targeted system is a chip multiprocessor having shared levels of cache. The applications executed on this architecture consist of sets of tasks that communicate through the memory hierarchy, thus through the shared cache. In the next subsection we present the available cache partitioning options.

### 3.1. Cache partitioning options

In the organization of conventional, set associative cache the address splits in three parts: tag, index and offset [4]. 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. The offset part of the address selects the desired word in the cache block. With respect to conventional cache organization we identify three possible types of partitioning:

- Associativity based, also called column caching [2] (Figure 1, a). In this situation a task gets a number of ways from every set of the cache. 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. However, the number of cache ways (cache organization) limits the granularity of the partitioning. We note here that in a large L2 cache, the state-of-the-art number of ways is around 16. Hence, if we have more than 16 tasks, some cache ways should be shared among them, leading to the already presented inter-task flushing (so compositionality) problem. Moreover, low partitioning

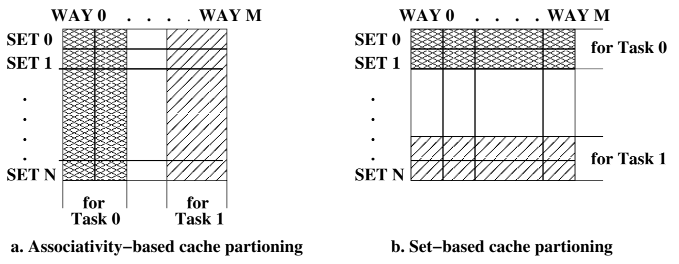

**Figure 1. Types of cache partitioning**

granularity limits the options of improving performance by tuning the partitioning ratio to the tasks requirements.

- Set based [6] (Figure 1, b). In this situation a task gets a number of sets from the cache. This type of partitioning requires translating the index such that it addresses another part of the cache as it originally did. This translation makes the set based partitioning more expensive, but due to the fact that typically in a cache there are more sets than ways, this methods can potentially induce compositionality.

- Mixed partitioning - is a combination of the first two. Every task gets a limited number of ways from a part of the cache sets.

### 3.2. Mixed cache partitioning

Three types of parallelism are possible in multimedia applications: functional parallelism (where task 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".

On one hand, a common method to achieve performance compositionality is by allocating to each task its own exclusive cache part. Therefore, multiple copies of a common region reside in cache causing coherence problems and having a negative impact on cache utilization. On the other hand, if the system has a shared cache partition for every common region, its compositionality cannot be achieved due to the cache flushing among tasks in the common cache.

To solve the problem of compositionality we first ensure that the instances of private tasks data and common regions don't trash each other in cache. Set based cache partitioning among each task and each common region guarantees this isolation. 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

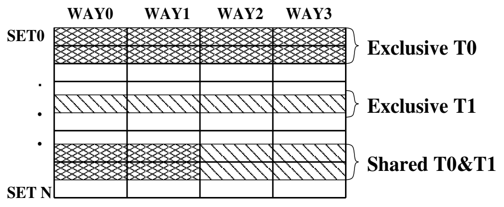

**Figure 2. Mixed cache partitioning**

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 data (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 using associativity based partitioning for the shared regions cache.

In conclusion, for achieving performance compositionality we use mixed cache partitioning like depicted in Figure 2. The dark grey 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 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.

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.

## 4. Mixed cache partitioning implementation

The envisaged architecture is the CAKE platform [15]. This platform consists of a homogeneous network of computing tiles (like the one in Figure 3) 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, fa-

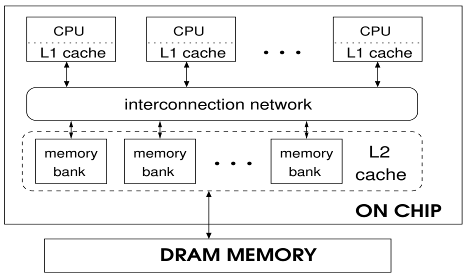

**Figure 3. Multiprocessor target architecture**

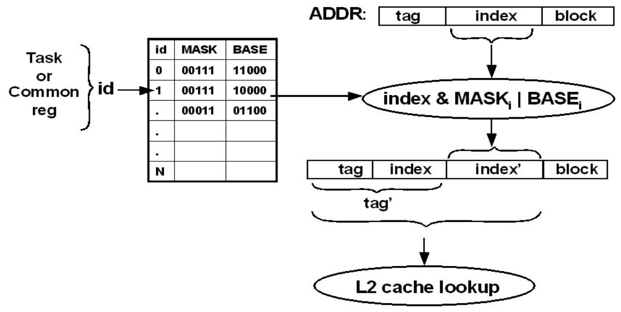

**Figure 4. Set based cache partitioning**

cilitating 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.

We apply the mixed partitioning on the L2 shared cache, because it is the most affected by the inter-task run-time conflicts. For the present work we assumes the followings: (1) the communication resources (busses, networks, etc.) are large enough (so the resource contention there is low) or they are also managed for performance compositionality, (2) since the levels of cache private to each processor are usually small and task switching rate in multimedia application is typically low enough, the L1 cache can be considered private to each task. In the following we present the implementation issues first for set based partitioning and then for associativity based partitioning.

As already mentioned, in a conventional set associative cache organization the address splits into three parts: tag, index and offset. The set based cache partitioning is done by translating the old index of an address into a new index before cache lookup (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.

The set based cache partitioning is done per task or common region instance so each memory access should be labeled with a *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 labelling the L2 accesses coming from the L1 instruction cache as code. At compile time it is known which tasks are instantiated multiple times so the code accesses of those tasks go into the same cache partition.

The associativity based partitioning is implemented by changing the cache replacement policy in case of a miss [13]. 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. Given that we provided 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.

In the existing light-weighted operating system responsible with task scheduling we added primitives for loading and modifying the necessary tables and registering address ranges of common regions.

## 5. Experimental results

For our experiments we used a CAKE multiprocessor platform [15] with 4 Trimedia processor cores running at 300 MHz and 4 ways associative L2 shared cache. The access times the different memory levels are as follows: to the L1 cache 3 cycles, to the L2 cache 12 cycles, to the off-chip memory 110 cycles. 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 [14]. 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 [3].

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 [8]. 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 task 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. To investigate the compositionality induced by cache partitioning we estimate the misses' variation in the same external conditions, so we used the same set of input data. For both application the misses' variation are smaller than 4%, so we can conclude that compositionality is achieved within reasonable bounds. The 4% 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 [1]. In Figures 5, respectively 6, the average miss rate and completion time for the two studied cache configurations are presented for the PiPTV and H.264 applications.

For the considered L2 sizes, the partitioned cache generally outperforms the shared cache. The relative miss rate reduction is, on average over the experimented cache sizes, 23% for the H.264 application and 25% for the PiPTV application. The typical L2 sizes for the CAKE platform are around 1-2 MBytes [15]. For this size the reductions in absolute miss rate are as follows: 3% for H.264 and 2% for PiPTV. This miss rate reductions results in a off-chip traffic reduction of 15% on average over the two applications, and in an execution time improvement of 5% for the H.264, respectively 3% 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). 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 critical path in the application gives the overall completion time, which is not the sum of tasks execution times. This can be observed in the case of the H.264 decoder running with 512 KBytes of L2 cache or in the case of the PiPTV applica-

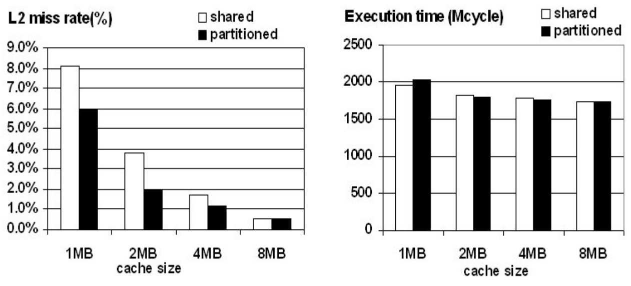

**Figure 5. PiPTV: shared vs. partitioned cache**

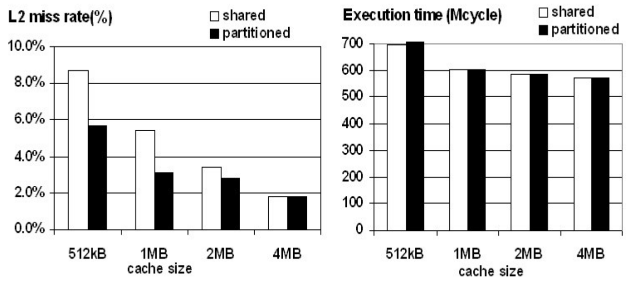

**Figure 6. H.264: shared vs. partitioned cache**

tion running with 1MByte of L2. Compared to the shared cache, the partition cache had a 3% lower miss rate but the execution time increased with 4%.

## 6. 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 an 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 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 4%, 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 on average to 15% reduction in the amount of off-chip traffic. Future work includes dynamic repartitioning strategies.

## References

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

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

[3] E. A. de Kock and all. Yapi: application modeling for signal processing systems. *Proceedings, 37th conference on Design Automation*, pages 402 – 405, 2000.

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

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

[6] D. B. Kirk. Smart (strategic memory allocation for real-time) cache design. *IEEE symposium on Real Time Systems*, pages 229–237, 1989.

[7] J. Liedtke, H. Härtig, and M. Hohmuth. Os-controlled cache predictability for real-time systems. *3rd IEEE Real-Time Technology and Applications Symposium*, June 1997.

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

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

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

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

[12] F. Sebek. The state of the art in cache memories and real-time systems. (01/37), Oct. 2 2001.

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

[14] E. B. van der Tol, E. G. Jaspers, and R. H. Gelderblom. Mapping of H.264 decoding on a multiprocessor architecture. In *Image and Video Communications and Processing 2003*, pages 707–718, May 2003.

[15] J. T. van Eijndhoven, J. Hoogerbrugge, M. Jayram, P. Stravers, and A. Terechko. *Chapter: Cache-Coherent Heterogeneous Multiprocessing as Basis for Streaming Applications, in Dynamic and robust streaming between connected CE-devices.* Kluwer Academic Publishers, 2005.