Editorial Manager(tm) for The Journal of Signal Processing Systems
Manuscript Draft

Manuscript Number: VLSI257R2

Title: Design and Tool Flow of Multimedia MPSoC Platforms

Article Type: Sp.Iss: MULTICORE Enabled Multimedia

Keywords: MPSoC; parallelization; multimedia; predictability; composability; scalability; tool flow; platforms

Corresponding Author: Dr. Bjorn De Sutter, Ph.D.

Corresponding Author's Institution: Ghent University

First Author: Bjorn De Sutter

Order of Authors: Bjorn De Sutter; Diederik Verkest; Jean-Yves Mignolet; Erik Brockmeyer; Eric Delfosse; Arnout Vandecappelle

Manuscript Region of Origin: BELGIUM

Abstract: This paper surveys components that are useful to build programmable, predictable, composable, and scalable multiprocessor-system-on-a-chip (MPSoC) multimedia platforms that can deliver high performance at high power-efficiency. A design-time tool flow is proposed to exploit all forms of parallelism on such platforms. As a first proof of concept, the flow is used to parallelize a relatively simple video standard on a platform consisting of off-the-shelf components. As a second proof of concept, we present the design of a high-performance platform with state-of-the-art components. This platform targets real-time H.264 high-definition video encoding at an estimated power consumption of 700mW.

*** Response to Reviewer Comments**

We rewrote the abstract as asked.

Design and Tool Flow of Multimedia MPSoC Platforms

Bjorn De Sutter
Ghent University, Belgium

Diedrik Verkest, Erik Brockmeyer, Eric Delfosse, Arnout Vandecappelle and Jean-Yves Mignolet
Interuniversity Micro-Electronics Center (IMEC), Belgium

Abstract. This paper surveys components that are useful to build programmable, predictable, composable, and scalable multiprocessor-system-on-a-chip (MPSoC) multimedia platforms that can deliver high performance at high power-efficiency. A design-time tool flow is proposed to exploit all forms of parallelism on such platforms. As a first proof of concept, the flow is used to parallelize a relatively simple video standard on a platform consisting of off-the-shelf components. As a second proof of concept, we present the design of a high-performance platform with state-of-the-art components. This platform targets real-time H.264 high-definition video encoding at an estimated power consumption of 700mW.

Keywords: MPSoC, parallelization, multimedia, predictability, tool flow

1. Introduction

Multimedia applications require ever more computation performance. At the same time, seamless mobility demands battery-operated devices that have to run with limited power resources, and that have to run multiple standards. A typical example of multimedia usage in the not too distant future consists of a video call between two people wearing cell phones. As one of them enters his living room, he continues the call on his HD multimedia device (what once used to be known as television), expecting high-quality video on that screen.

Not only does the platform need to deliver all this functionality at run time, it also needs to meet several design constraints. For enabling a short time-to-market, the platform should be as flexible as possible, meaning programmable, and the components should be reusable in a large range of products that should also be cheap to produce. Thus the platform design itself and the programming of it should be flexible.

In the past single processors with increasing clock frequencies met the demands for more computing power, enabling programmers to keep using their sequential programming. Today, and certainly in the near future, simply increasing clock frequencies to get higher performance is no longer a viable option. Today lower frequencies are more attractive from the power-efficiency point of view, so lower clocked multi-core

processors on a chip (MPSoC) are required. However, to exploit the potential of MPSoC solutions, the application or algorithm development process needs significant change.

To start with, all types of parallelism, being instruction-level parallelism (ILP), data-level parallelism (DLP), and thread-level parallelism (TLP), need to be extracted from applications and then mapped onto the parallel computation resources of the platform. As different parts of the applications can exploit different kinds of parallel computing resources, the ideal platform is heterogeneous, offering different kinds of processing cores: reduced instruction set computers (RISCs), digital signal processors (DSPs), field programmable gate arrays (FPGAs), application specific instruction-set processor (ASIP) accelerators, etc. Furthermore, for limiting the power consumption to a minimum while still allowing the parallel code to access enough data in parallel to achieve high throughput, a complex memory hierarchy needs to be implemented in hardware and supported in software. This means that all data transfers in the application need to be determined, and that they also need to be mapped onto the available memories and data transfer primitives offered by the run-time libraries of the platform.

So compared to single-core systems, new hardware is needed to make multiple cores operate in parallel, and new programming aids needs to help in transforming the software to exploit that hardware. For a platform or a range of platforms to succeed, we believe it is crucial that the new design flow is to a large extent automated. Besides the existing, well-known support from single-core compilers, additional automated tools are needed to support the mapping of TLP onto heterogeneous computing resources, and to support the mapping of data transfers onto communication primitives. The reason is obvious: neither of these tasks scales when performed manually.

This paper reports on the results of a recent multimedia MPSoC research project at IMEC (Interuniversity Micro-Electronics Center). The main contribution of this paper is to demonstrate that a largely automated tool flow can indeed be built by carefully choosing the hardware components and the way they are exposed to the application by the run-time library. We discuss a series of platform components of which concrete instances deliver the required performance at a high power-efficiency. Equally important, we discuss how these components are abstracted by the run-time library at a level of predictability that allows both engineers and tools to deal with the platform's complexity and real-time behavior. Based on this predictability, we then present a tool flow that maps complex applications onto the proposed components.

To demonstrate the effectiveness of both the platforms and the tool flow, the achievements of two experiments are reported. Due to the

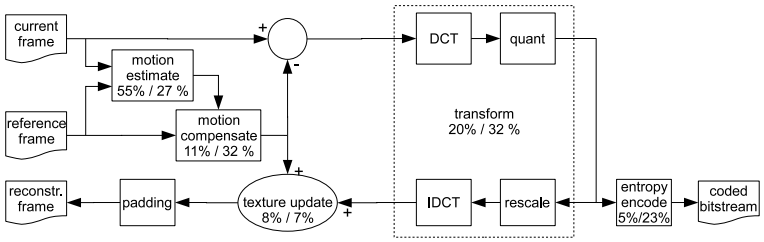

Figure 1. An exemplary codec. For two program versions, each kernel's contribution to the workload is presented: first for unoptimized code executed on a VLIW, and secondly for DLP-optimized code executed on an ADRES (see Section 3.1).

limited number of resources at our research institute, neither of the experiments result in a production ready design. Together, however, we believe they provide a valid proof-of-concept case study of the proposed platform template and tool flow. In a first experiment, a video decoder (QSDPCM) is mapped to a platform built with off-the-shelf components. This experiment demonstrates that we indeed have a complete tool flow to map full applications onto platform. In a second experiment, we have designed a state-of-the-art platform that supports more demanding applications (MPEG-4 encoding and decoding), using in-house state-of-the-art components, that demonstrates that system instances of the proposed template can indeed be built that deliver the performance and power-efficiency required for near-future applications.

This paper is structured as follows. Section 2 discusses components that are suitable for MPSoC multimedia platforms. Section 3 discusses the tool flow to map applications onto those platforms. Section 4 reports on the two experiments, and conclusions are drawn in Section 5.

2. Multimedia Platform Components

This section presents IMEC's view on the types of components of which platforms should be built in order to deliver the high performance, high power-efficiency, and predictability at manageable levels of complexity.

2.1. Multimedia Application Opportunities

Typically, current state-of-the-art hybrid block-based video encoders (MPEG-2, MPEG-4, AVC) conform to block diagrams as shown in Figure 1, in which each block corresponds to one or more kernels (loops of computations) to be executed on data sets. The compression in multimedia codecs is achieved by exploiting temporal correlation in motion estimation/compensation (ME/MC), spatial correlation (discrete

cosine transform (DCT) and quantization) and statistical correlation (entropy coder). Furthermore, to avoid so-called *drifting* at the decoder or, in other words to match the encoder and decoder prediction signals, the encoder incorporates the same decoder reconstruction path (inverse quantization or rescaling, and inverse DCT). The resulting reconstructed frames are used to perform the temporal prediction for the next input frames. The encoding process is performed on macroblocks (MBs) of 16x16 pixels or 8x8 pixels, or even subblocks of 4x4 pixels, which form a partition of the input image (Richardson, 2003).

The performance requirements of such a video coder depend on the resolution and the frequency of the inputs, and on the rate-distortion requirements. To meet those a certain number of Operations-Per-Second (OPS) needs to be executed. In the case of AVC Baseline Profile encoding at HDTV 720p resolution at 30 frames-per-second, this minimum performance is in the order of 100 integer GigaOPS (GOPS), depending on specific encoder algorithmic implementations and settings (Chen, 2006; Huang, 2005; Pinto, 2006). The only way to deliver this and consume acceptable amounts of energy is through parallelization.

In the case of video codecs, the opportunities for high ILP are mainly limited to the inner loops that operate on MBs or subblocks. These can be software-pipelined to expose enough instructions that are not dependent on each other, and can thus be executed in parallel. Non-loop code, involving many (unpredictable) control flow transfers, does not offer many ILP opportunities. Because the inner loops work on small datasets (MBs), their number of iterations is usually limited, and so is the amount of ILP that can be exploited. For a similar reason, the amount of available DLP is limited. In order to be really worthwhile single-instruction multiple-data (SIMD) or vectorization needs the data words that are loaded from memory to be stored consecutively; otherwise the memory accesses cannot be executed in parallel and many packing and unpacking instructions need to be inserted. As the inner loops are bound to MBs, typically only 4 to 16 pixels are stored consecutively. In practice, we believe that a single DSP core may support SIMD up to at most 4 parallel subwords, because of the aforementioned limitations, and that no more than 16 FUs can be utilized well. The latter is supported by our experiments with different ADRES DSPs (Cervero, 2007), of which some results are discussed in Section 3.1. This means that at most 64 operations per cycle can be executed on a single DSP core. Supposing such a core operates at a frequency of 400 MHz, which we were able to achieve for some of our ADRES DSPs, the peak performance is 25.6 GOPS. Hence to achieve anything in the order of 100 GOPS, at least 4 cores running one thread each in parallel are needed.

This multithreading can either be done in the form of a functional pipeline, in the form of a data parallel path (by, e.g., parallelizing the handling of one frame into two threads that each handle one half of the frame), or in any combination thereof. In block-based video coding, this parallelism can be achieved by exploiting the block-based processing of the input image. For example, we can perform the ME on multiple MBs simultaneously or we can perform the DCT on one MB while performing the ME on the next one. The difficulty is to identify the data dependencies between different kernels enabling such a partitioning into well-balanced threads with as little overhead as possible. Looking at the computational costs of the components in Figure 1, it is clear that a straightforward application of pipelining parallelism alone will not suffice. Of the optimized kernels about one third of the work is spent in MC, so pipelining alone can certainly not deliver a four-fold speedup.

2.2. Processors & Platform Components

Different parts of codecs expose different kinds of parallelism. To exploit these power-efficiently, multiple heterogeneous computational resources are needed. To store and transfer the data on which the computations are performed, storage and interconnect resources are needed.

2.2.1. *Computational Resources*

Many types of computational resources exist. The most power-efficient resources are application-specific integrated circuits (ASICs). While these are not flexible, being not programmable, they may be very useful to execute specific tasks at very high energy efficiency. Typically, ASICs are added to platforms as accelerators for such specific tasks.

Compared to ASICs, FPGAs are very flexible, but unfortunately they are too energy-inefficient and too costly for consumer products. Furthermore, only code that is programmed in spatial computational models, such as structural VHDL, is easily mapped onto FPGAs. C code, which exposes a much more temporal model, can still not be mapped onto FPGAs efficiently, with the exception of inner loops that can be transformed, by compilers, into very spatial-like representations.

General-purpose processors (GPPs) are not optimized to any particular type of parallelism, and hence they are by definition not power-efficient for any of the specific parts of a complex embedded application. Except of course for those parts where little parallelism is to be found, such as in the control-flow intensive parts of applications that basically implement state-machines. For such parts, GPPs that focus on power-efficiency rather than on pure performance are interesting.

What remains is different types of DSP architectures and ASIPs. Because of their good balance between power-efficiency and compiler support, a broad range of very long instruction word (VLIW) architectures exists today, ranging from high-performance Texas Instruments DSPs (http://www.ti.com), over SiliconHive (Halfhill, 2003), to very-low-power architectures such as CoolFlux (www.coolfluxdsp.com). Some architectures focus primarily on DLP (vector machines, but also SIMD extensions), while others focus primarily on ILP (such as ADRES, see Section 3.1), and still others combine both. One important problem caused by the huge diversity of computational resources in general, and of DSP architectures in general, is that application loop kernels need to be transformed differently depending on the target processor in order to exploit its features optimally (Franke et al., 2005). On top of the parallelization and memory hierarchy mapping task, this complicates the mapping of applications onto heterogeneous platforms even more.

As we focus on programmability, our target platforms may contain GPPs and DSPs, but ASIC accelerators are supported just as well.

2.2.2. *Storage and Communication Resources*

The data on which an application performs computations need to be stored in some form of data memory. Because of programmability, these memories should be generic. So in our platform, we will not consider hardware implementations of specific buffers, queues, etc.

In general, a memory's power consumption and delay increases when its size or its number of ports grows. These relations are superlinear, which implies that we should try to improve power consumption and possibly performance by building systems with as small as possible memories. Still, large data sets need to be stored, and moreover, as we have many computational resources operating in parallel a lot of data needs to be accessed in parallel. For that reason, hierarchies of distributed storage resources are the only viable solution. An excellent overview of memory hierarchies and programming opportunities can be found in (Wehmeyer and Marwedel, 2006).

One important question to answer is whether we use scratch-pad memories (SPMs) or caches, or a combination of the two. We strongly favor SPMs because of their superior power-efficiency, scalability, predictability and composability

With respect to *power-efficiency*, SPMs are smaller and more efficient per access, and this difference becomes even bigger in the case of MPSoC systems where cache-coherency is a big (and expensive) issue.

Scalability in this context refers to two things. First it refers to the fact that the performance of a platform should scale with added resources. Secondly it refers to the fact that a platform's predictabil-

ity and composability should not be endangered by adding more resources. When a number of original applications have been mapped to a platform exploiting the predictability and composability of that platform, extensions to that platform or additional applications should not break the original applications. In this respect snooping-based cache coherency is not scalable because every processor is putting all its writes directly on the bus, protocol-based cache coherency has a large latency, and software cache coherency requires a user/tool to add explicit flushes and invalidates. While the latter can exploit buses much better, it is limited to buses and bus snooping, and hence not nearly as scalable as network-on-chip based solutions.

Predictability is primarily a design-time notion. Predictability is required to enable designers and tools to reason about how an application uses storage and communication resources, and hence to estimate the needed amount of resources for a certain application to reach a certain performance. For example, predictable storage and communication resources allow a tool to calculate how long a certain data transfer will take, which is necessary to compute the communication overhead in a mapped application in terms of execution cycles. *Composability* is more a run-time notion. For guaranteeing that an application can deliver a certain amount of performance, a certain amount of guaranteed communication and storage resources should be available to that application. Composability in this context refers to the ability of a run-time manager to decide how many applications can run concurrently while still guaranteeing that all of the running applications get the resources they were assumed to have at their disposal at design time. Obviously composability requires predictability, but predictability also depends on composability: when there is no composability, an application cannot be guaranteed to have a certain amount of resources, and hence its behavior becomes unpredictable.

Good SPM management requires analysis and optimization at design time, and support for data transfers, a.k.a. block transfers (BTs), at run time. When done well, the design-time analysis and run-time SPM management (SPMM) can offer some important advantages over caches. Foremost, using profile data a design-time analysis can take into account a whole program execution. So it can introduce actions at program points that anticipate future run-time behavior. Purely dynamic analysis, such as present in caches, can only analyze past behavior during a program execution. Furthermore, SPMM can more accurately copy exactly that part of the data that is needed by some core, such as by reading only a single column of bytes from a frame. And data that is "written only" by some processor does not need to be fetched first. This contrasts with caches, that fetch a line as soon as one byte

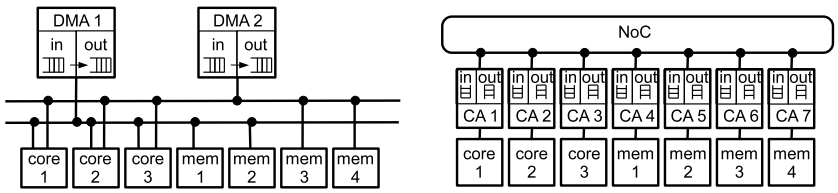

Figure 2. On the left, a traditional interconnect with buses and DMA controllers. On the right, a more modern interconnect consisting of CAs and a NoC.

in it is written. Moreover, many arrays can be stored in L1 SPM and do not need to be fetched or flushed when replaced by something else. Note that caches can also support locking. But to enable this locking, a similar, albeit maybe simpler, design-time analysis and optimization is required as for SPMs. Overall, a managed SPM hierarchy is more predictable, and hence more composable than a cache-based system.

For similar reasons, we propose to use communication assists (CAs) and a network-on-chip (NoC) communication rather than buses and direct-memory-access (DMA) controllers. A CA is essentially a distributed DMA. Each CA contains half a DMA for sending and half a DMA for receiving. This allows two CAs to setup a DMA transfer by using a specific BT protocol. Figure 2 depicts a DMA-bus interconnect on the left, and a CA-NoC on the right. Conceptually, a DMA controller can be seen as two address generator units. One computes in-addresses to fetch data from some component, while the other computes out-addresses to store data into some other component. Inside the DMA controller, fetched data flows from the input-buffer to the output-buffer to be written again. When a DMA controller is active, it occupies the bus to which it is connected, so that bus cannot be used for anything else. The CA-NoC operates differently. At the sending end the CA will set up sending through its output FIFO and at the receiving end the CA will set up the receiving through its input FIFO. When a NoC is used, many CAs can be communicating with each other over point-to-point connections implemented over the NoC. This CA-NoC scales much better than the traditional DMA-bus setup. Furthermore, when the NoC offers Quality-of-Service (QoS) protocols, the whole communication becomes predictable and composable, as the run-time manager can set up point-to-point connections with guaranteed bandwidth and latency. The alternative is best-effort communication, which implies overdesigned communication resources, and hence inefficient resource use. For more information about NoCs and CAs, we refer to (Benini and De Micheli, 2006; Goossens et al., 2002; Goossens et al., 2005).

3. Application Mapping Tool Flow

This section focuses on tools and techniques that support the (automated) mapping of sequential application functionality to parallel MPSoC platforms, exploiting all levels of parallelism.[1] The next section will then present a mapping flow based on these tools and techniques.

3.1. Tools for Extracting Instruction-Level Parallelism

In this section we focus on VLIW DSPs and alikes. Typically, those cores execute the code with high ILP potential. Control code, in which ILP is limited by control and data dependencies, is typically mapped onto more sequential processors, such as an ARM.

Many techniques have been developed to expose as much as possible the available ILP in application loop kernels (where most of the computation time is spent), and to map that ILP onto the parallel issue slots of VLIW processors. The formation of hyperblocks (Mahlke et al., 1992) by means of predicated execution (in which control flow is replaced by data flow) and the application of software-pipelining by means of modulo scheduling (Rau, 1995; Lam, 1988) are among the most commonly used techniques to overcome limitations on ILP that are caused by data and control dependencies. When these techniques are applied on top of traditional data flow and control flow optimizations in a compiler, reasonable levels of ILP can be exploited, and commercial tools are available that automatically apply these transformations and generate code for (clustered) VLIW architectures of up to 8 issue slots. However, when the number of FUs increases to 16 or so, which does make sense for a lot of application loop kernels, additional novel compiler techniques are needed. To let that many FUs operate with minimal power consumption, their interconnect to the register files must become much sparser (to limit the number of ports per register file), up to the point where traditional register allocators and list-scheduling based schedulers do not perform anymore. So not only do we need architectures with fewer interconnects, but we also need new compilation techniques for those architectures.

Several architectures and corresponding compiler techniques have emerged over the last years, operating at different performance/power specifications. Architectures such as EDGE (Burger et al., 2004) combine dynamic instruction issuing with static instruction placement (i.e.

[1] In the future application development might be done in parallel languages instead of sequential C. However, this will not solve the mapping problem. Rather than distributing operations over several processors as needs to be done now, the problem will then become to cluster and assign operations to the limited number of processors. Only the design flow will then be different, not the fundamental problem.

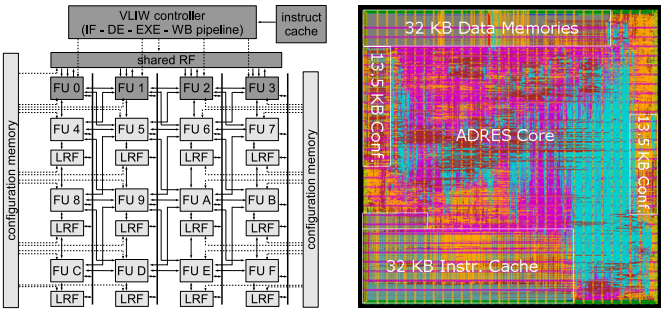

Figure 3. An ADRES architecture consisting of an array of 16 functional units that are connected with almost as many local register files, and a layout of a multimedia version of this architecture.

compile-time assignment of operations to issue slots) on an array of functional units, aiming for general-purpose high-performance applications. An architecture template with high ILP potential at low power consumption is ADRES (Mei et al., 2003a). Instances of this template consist of an array of functional units that are connected to many small local register files (RFs) in a sparse interconnect, and with one central RF. The whole array executes loops in a data flow mode for which the instructions and interconnect configuration (multiplexer selection bits) are fetched every cycle from a configuration memory. As such the array operates as a kind of dynamically reconfigurable coarse-grained reconfigurable array, i.e. as an FPGA with wide computation resources (32 or 64 bit instead of 1 bit) that is reconfigured every cycle. The top row can operate as a regular VLIW processor, with a standard I-cache and fetch-decode-execute-writeback pipeline, to execute non-loop code. Data passing from non-loop code to loop-code happens through the central register file, that is shared by both modes, and by the shared memory, that is accessed through the load/store units that are shared by both modes. An example ADRES instance with 16 functional units is depicted in Figure 3.

For the ADRES architecture template, a full ANSI-C compiler tool chain exists (including simulators, virtual machines, binutils, and a VHDL generator) that compiles inner loops in applications for the ADRES array mode, and all other code to VLIW mode. This compiler basically builds a data dependence graph representation for the loops, and maps this graph on a time-space graph that models all interconnects in the ADRES array. This compilation technique is similar to placement and routing techniques from the hardware synthesis world (Betz et al., 1999). On top of the time-space graph modeling of

H264 AVC decoder kernel	IPC
macroblock filter 1	11.9
macroblock filter 2	9.9
macroblock filter 3	10.0
macroblock filter 4	11.7
motion compensation	10.0
find frame end	3.9
IDCT 1	12.0
macroblock filter 5	12.4
C-library memset()	5.0
IDCT 2	12.7
average	**10.0**

MPEG-4 encoder kernel	IPC
motion estimation 1	12.5
motion estimation 2	12.0
texture coding 1	10.4
MBSAD computation	12.0
texture coding 2	4.5
texture coding 3	13.0
texture coding 4	13.0
texture coding 5	13.7
texture coding 6	13.0
motion estimation 3	13.0
average	**11.7**

Figure 4. IPCs obtained on a 16-FU ADRES DSP for the ten most time-consuming kernels of an H264 AVC baseline profile decoder and of an MPEG-4 encoder.

the architecture, a modulo-reservation table is used to ensure that loop code is modulo-scheduled, i.e., software-pipelined. For more details on this algorithm, we refer to (Mei et al., 2003b).

The result of a synthesized ADRES instance after layout is depicted on the right of Figure 3. This experiment was performed using 90 nm standard cell TSMC logic. The total area is $3.6mm^2$, and the clock frequency obtained is 300 MHz at which 91 mW is consumed when an AVC video is being decoded. For real-time AVC BP CIF decoding at 30 fps, a 50 MHz clock speed suffices, meaning that 15 mW is consumed. For D1 decoding, a 205 MHz clock and 62 mW suffice. Please note that this is an architecture with 16 FUs and 12 small local RFs, all of which are 32 bits wide. Special instructions, that are programmed via intrinsics, include clipping, min/max operations, and 2-way SIMD.

Some compilation results for an AVC decoder and an MPEG-4 encoder are presented in Figure 4. For most loops of the 10 hottest loops of each application, the number of instructions executed per cycle (IPC) is well above 10. When the IPC is lower than that, this is always due to data dependences between operations in consecutive iterations of the loops, which form a fundamental limitation on the obtainable IPC. More extensive results are presented in (Mei et al., 2008). These results illustrate that high instruction-level parallelism can indeed be obtained for compiled C code, at low power consumption.

3.2. Tools for Extracting Data-Level Parallelism

DLP can be exploited in many ways, from completely manually to fully automated. Compiler support for extracting vector operations from an application has been researched for many years now, and even the popular GCC 4.3 compiler includes rather extensive autovectorization support (http://gcc.gnu.org), in the form of autovectorization analyses and code transformations, but also in the form of vector data types (that extend the standard C data types) that allow a programmer to

express vector operations in his code. Another popular, albeit manual, technique to program SIMD operations is the use of intrinsics: in C-code, the parallel operations are programmed by means of function calls to functions that take wide inputs (e.g., of 64-bit `long long` data that contains 4 16-bit values) and produce a wide output. When the compiler sees the function calls to these functions, they are simply replaced with single, corresponding instructions on the target architecture. This way, the compiler does not need to do instruction selection for the intrinsic operations, which may be very complex and hence difficult to use automatically. In our tool flows, intrinsics are used both for programming SIMD and for programming other complex, specialized instructions that were added to the ADRES architectures for speeding up multimedia applications, such as clipping instructions.

3.3. Tools for Extracting Thread-Level Parallelism

With respect to TLP, we should first note that manual parallelization is inherently non-scalable because of the retargeting effort that is required to map applications onto different platforms. Thus, we need to find (semi) automated ways to parallelize applications. With today's technology, mapping an application's TLP such that resources are used efficiently and correctness is guaranteed remains a big challenge, even if applications offer plenty of TLP potential. Fortunately, the behavior and workload of multimedia applications are rather predictable at design time as they consist of a largely (but not completely!) fixed execution order of a number of kernels like the ones in Figures 1 or 4.

The main task in exploiting TLP is deciding on the *distribution* of kernels: which kernel runs on which processor. From a specific distribution, it is possible to derive the communication and synchronization required to maintain correctness. These, in turn, determine how efficiently resources are used. Since the application workload is relatively predictable, it is possible to take most decisions at design time, thereby avoiding the overhead of an operating system and of switching. In this paper, we discuss an approach which is *fully* design time, i.e. all decisions (workload distribution, processor assignment, communication) are taken at design time. Instead of an operating system, a simple run-time library executes the design-time decisions. These decisions do not involve scheduling: all the kernels assigned to a processor can simply be executed in the same order as in the original sequential code.

In our semi-automated approach, a developer iterates through a parallelization space of an application, assisted by two types of tools. In each iteration, the developer (1) specifies how the kernels should be distributed, (2) uses a tool to automatically generate correct parallel

code for the specified distribution, and (3) uses other tools to assess the efficiency of the performed parallelization. Based on that assessment, the developer goes back to step 1, updating his specification of the distribution until an acceptable efficiency is obtained. *Nowhere in this process is it necessary to write parallel code manually.*

As discussed in Section 2.1, two main distribution mechanisms exist: functional parallelization, which distributes different kernels over different processors, and loop or data parallelization, which distributes different iterations of the same kernel to different processors. The freedom for such parallelization is limited by data dependencies. In multimedia applications, the dependencies typically still allow a very large number of different parallelizations because even if a dependency exists between two kernels or two executions of the same kernel, they can still be executed in parallel by pipelining over a surrounding loop.

When a dependency exists between two kernels executing on different processors, this implies that data has to be communicated from one processor to the other, and that the involved processors must synchronize: the consuming kernel can only consume data when it is ready, and the producing kernel should not overwrite data before it has been consumed. In many multimedia applications, as in streaming applications in general, the easiest way to implement both the communication and the synchronization at the same time is by using a FIFO. This makes the synchronization one-way, and allows the producing kernel to continue without waiting for the consuming kernel, because the data is buffered in the FIFO.

Finding the dependencies is easy enough for scalar data using Static Single Assignment analysis (Muchnick, 1997). For non-scalar data (arrays and pointer structures) dependencies, more accurate analyses are needed that identify exactly which parts of the non-scalar data are being produced and consumed (Pugh and Wonnacott, 1998; Feautrier, 1991; Plevyak et al., 1993), and hence which data need to be communicated. Alternatively, the non-scalar data can be stored in shared memory and only a counter or pointer is passed to the consumer that indicates which data is ready for consumption. With this method, however, output dependencies and anti-dependencies still need to be handled explicitly by means of additional counters and pointers. Otherwise data may be overwritten before it is consumed.

We have implemented a tool that automates the generation of multithreaded code (Cockx et al., 2007). Its input consists of sequential C source code that has been annotated by the developer with information (in the form of pragmas) on how he wants to apply functional and data parallelization. This tool generates correct parallel code by identifying all required FIFOs or shared memory synchronization using the

aforementioned dependency analyses. Concretely, the tool duplicates, for each processor, the kernel code and surrounding control code from the sequential code, after which the required FIFO and synchronization statements are inserted. This tool guarantees correct parallelization by construction. However, this tool by itself does not suffice to perform a parallelization. It merely generates correct parallel code, but not necessarily efficient parallel code. To obtain an efficient parallelization, the parallelization specified in the developer's annotation should solve the following three problems.

Load balancing issues occur if one core has less work to do than others. As a result, it will waste resources remaining idle for some time. Obviously, when some kernels do not have a fixed execution time, design-time load balancing becomes difficult. Still, it is usually possible to find a distribution that balances execution time variations over a longer period.

Synchronization issues result from dependencies. The reader of some data cannot start until the writer has finished. Worst case, dependencies force a completely sequential execution order. After inserting the synchronization mechanisms (FIFOs) in the code, it is possible to determine the time consumed by the dependencies either by simulating the parallel code or by formal analysis (Denolf et al., 2007; Burns et al., 1995; Baccelli et al., 1992).

Communication issues arise when communication resources are being shared. Usually, the platform does not offer a fully connected point-to-point communication network. The communication resource is to some extent shared between processors. This is particularly the case for shared memory. On a single processor, a communication takes a predictable amount of time, but on the multiprocessor it may take longer because it has to wait for the request of another processor to finish. It turns out that for some latency-sensitive communications, the additional waiting time becomes critical for the overall execution time (Marescaux et al., 2007). Although the distribution decision affects the communication overhead quite a lot, it is less important to take communication into account for the distribution decision. Indeed, it is possible to explicitly manage data communication and move most transfers to a place where their latency is not critical (Dasygenis et al., 2006; Marescaux et al., 2007).

We have developed a simulation tool that allows the developer to assess the efficiency of a distribution without requiring a time-consuming full mapping and simulation on the parallel platform. This high-level parallel simulation environment is based on information from a sequential profiling run. The sequential run is performed on the target processor (or a simulator of it) and records the execution time of each

Figure 5. Screenshots of the waveform representations of a program execution. For two ADRES cores, the activities related to BT synchronization, BT issuing and actual computation are shown. From the zoomed -out shot , two stages can be observed: one in which the ADRES processors are active, and one in which they are not, i.e. when the ARM processor on the platform is actually executing. From the circled areas in the zoomed-in shot, it can be observed that the two ADRES cores are effectively performing computations in parallel.

kernel. These times are used in the parallel run on the workstation to simulate the synchronization overhead. Thread activations, FIFO filling and waiting times are all recorded and presented in two formats. First, computation times and communication times can be displayed in convenient graphs such as the ones displayed in Figure 9 (see Section 4.1 for an explanation of the graphs). Secondly, they can be displayed as waveforms, as shown in the screenshots in Figure 5. These two formats give a designer a quick view on the potential issues of his parallel application's timing with and without the actual variations in execution time, which allows him to perform an efficient exploration.

The most related work is OpenMP (Chandra et al., 2001). Like our tool, OpenMP is a compiler extension that enables the parallelization of C code by means of pragmas. It is supported by its own run-time library, and is great for specifying data parallelization of loops with independent iterations. For other forms of parallelism the user has to take care of data dependencies and he has to rewrite the code before inserting the pragmas. For example, for functional pipelining, the developer has to implement the activation and deactivation of pipeline stages in C code, thus basically implementing the whole pipelining himself. The pragmas then only instruct the compiler to execute the stage in parallel. Our tool also relies on pragmas, but no C code rewriting is necessary, which is the key to facilitating design space exploration.

3.4. Tools for Managing Scratch-Pad Memories

To avoid a power and performance bottleneck in main memory accesses, it is often useful to give threads local copies of data that can be accessed in local memories. Obviously, the energy and bandwidth consumed in making copies and transferring blocks of data between different mem-

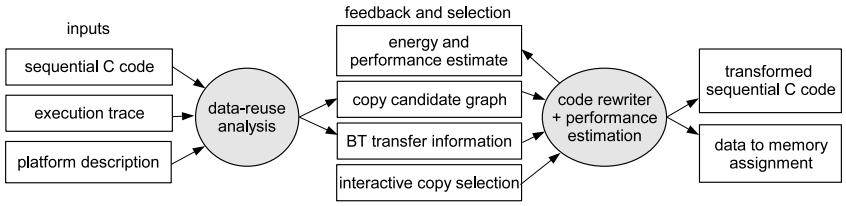

Figure 6. Inputs and outputs of our tool to insert data block transfers.

ories should be balanced with the gain of avoiding global accesses. To enable the developer to find a good balance, we have developed another tool that automates a lot of the code rewriting involved in inserting data copies and data block transfers. Like the tool for inserting TLP in the previous section, this tool enables the developer to explore different options without having to rewrite C code manually. In other words, this tool enable our flow to scale to larger platforms and applications.

The main inputs and outputs of this tool are depicted in Figure 6. Based on a description of the platform and a profile of the application, a data-reuse analysis learns how data is accessed and reused by the kernels of the application. This analysis highlights good candidates for local copies in the copy-candidate graph, and informs the developer about the required block transfers. The developer can then choose which local copies he wants to see implemented, after which the tool will give him an energy and performance estimate, a data assignment to local memories, and rewritten C code that includes the calls to the run-time library to execute the necessary block transfers.

Existing other work in this area is presented in (Kandemir and Choudhary, 2002; Udayakumaran and Barua, 2006; Udayakumaran et al., 2006; Verma, ; Wehmeyer and Marwedel, 2006; Aouad and Zendra, 2007). Overall, we believe that our tool is applicable to a much broader range of applications than the existing work, as is demonstrated on a full MPEG-4 encoder (Baert et al., 2008).

3.5. Supporting Run-time Library

To make sure that MPSoC platforms with components as described in Section 2 can be programmed efficiently using a design flow like ours, a *run-time library* is needed that abstracts the hardware. By means of different platform-dependent implementations, the run-time library offers a platform-independent interface to the hardware resources. A well designed run-time library interface carefully balances abstraction and performance. A more abstract interface hides more of the platform, which improves application portability. On the other hand, for performance reasons the interface should expose many platform features

which can be used to optimize the application for the platform. In our context, where we start from sequential code and make all decisions at design time, the following run-time functionality is required.

Regarding the platform's configuration, a *thread configuration* statically defines the threads, the functions representing the threads, which processors they run on, and other thread parameters. A *FIFO configuration* statically defines the FIFOs, which threads use them to communicate, their block size, and their depth. A *counter or pointer configuration* statically defines the counters or pointers (token FIFOs), by which threads they are used, their range and the allowed difference.

Concerning application-wide control flow, the assumption is that the outermost control flow is not parallelized. For that reason, a sequential *root thread* must be identified that starts the pre-defined parallel code at specific points. Once all pre-defined threads have run to completion, they perform a *barrier synchronization* before the sequential code is entered again. To support control flow that jumps out of parallel sections, as when an exception happens, special support for *thread termination* needs to be executed before the sequential code is entered again.

Regarding communication, the *FIFO communication primitive* `put` blocks if the FIFO is full. The `get` blocks if it is empty. It is also useful to split these primitives in an `acquire` and a `release` part such that the data can be written directly into the FIFO buffer rather than requiring an additional copy. The acquire is then blocking and returns a pointer into the FIFO buffer, while the release is non-blocking. *Counter increment and test* are the equivalents of token FIFOs. The increment is non-blocking while the test is blocking. For the communication of data blocks, *block transfers* are used. Rather than communicating individual words, it is more efficient to let a DMA unit transfer blocks of data.

Note that some typical primitives are not required, e.g. semaphores, locks, FIFO peeking, dynamic thread creation. This is so because we start from sequential code and take all decisions at design-time.

3.6. Overall MPSoC Mapping Flow for ILP, DLP and TLP

Based on the tools and techniques described in Section 3, as well as on the run-time library functionality, we can build an overall MPSoC mapping flow, as depicted in Figure 7.

Our flow relies on transformations of the source code. Ideally, the whole process would be automated and the programmer could just push sequential C code in a tool which spits out optimal TLP, DLP and ILP machine code. Unfortunately, manual interaction is still needed at every step to achieve good results. Thus, the code has to be easy to transform

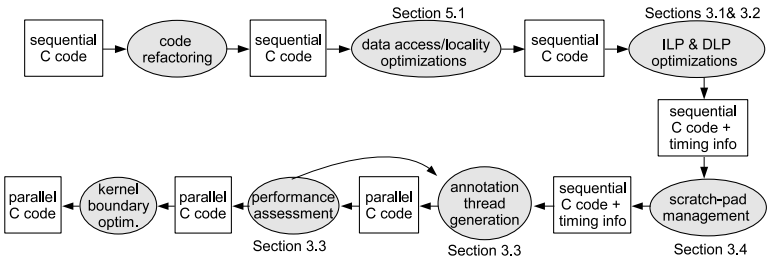

Figure 7. Our overall application mapping flow.

and to understand. This often requires refactoring of existing code to make it amenable for manual or automatic analysis and transformation.

Multimedia applications are typically data-intensive, involving many memory accesses and much communication. Optimizing temporal locality (Wolf and Lam, 1991) reduces the amount of communication required after parallelization, as demonstrated in Section 4.1. This can be done on the sequential code, and is mostly platform-independent. The locality optimizations are therefore also scalable to platform extensions.

To evaluate the quality of load balancing, it is important to have a good idea of the load generated by each kernel. For that reason, the code is first optimized for the ILP and DLP on a single processor. After parallelization, the code at the boundary between two kernels that are mapped onto the same processor can often still be optimized. Therefore, the final step in the flow refines the kernel optimizations.

In our current, mature flow, we apply the scratch-pad management strictly before thread-level parallelization. The reason is pragmatic: the scratch-pad management tool cannot deal with the more complex code that comes out of the parallelization tool. Merging the two tools is ongoing work, which we believe will lead to better results.

We should note that the parallelization into threads occurs fairly late in the mapping flow. This has the advantage of being more scalable: when mapping to a higher-performance platform which contains more processors, only the parallelization step has to be redone while the results of the locality and kernel optimizations stay valid.

4. Experimental Evaluation

This section presents results and lessons learned from two experiments: one in which the whole tool flow was applied on a full application for an off-the-shelf platform, and one in which we designed a state-of-the-art platform using the components discussed in this paper.

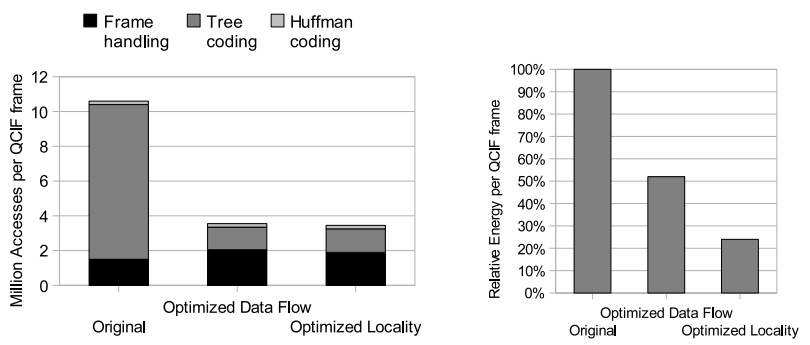

Figure 8. Number of accesses to large data structures in QSDPCM at various stages of data-flow and data locality optimization, and the corresponding energy estimations based on the used two-level memory hierarchy.

4.1. Mapping QSDPCM on an off-the-shelf platform

To demonstrate the effectiveness of our design flow, we mapped a simple multimedia application onto an off-the-shelf platform. Quad-tree Structured Difference Pulse Code Modulation (QSDPCM) is a typical, albeit simple, video codec. It consists of an inter-frame compression based on a three-stage hierarchical motion estimation (ME), which we split in two kernels, followed by a quad-tree based coding of the motion-compensated (MC) frame-to-frame difference signal, and a quantization (QC) followed by Huffman compression. The latter two steps are combined into one kernel. The target multi-processor platform consists of 6 TI-C62 DSP processors with local L1 scratch-pad memories, the Aetheral network-on-chip, a shared L2 memory, and CA blocks. A cycle-accurate virtual platform model of the architecture in SystemC was used to obtain accurate performance estimates. The remainder of this section discusses a selected number of steps out of the design flow.

The first step we illustrate is the optimization of data flow and data locality by performing loop transformations. These reduce the number of accesses to large arrays and improve their locality by reordering loop iterations. Figure 8 shows their effects on the number of memory accesses performed and on the energy consumption per frame. Note that the locality-improving transformations do almost not affect the number of accesses to data. They only affect the access locality of the data, thus reducing the number of power-consuming accesses (or data block transfers) to and from L2 memory.

The next step explores different thread-level parallelizations. Figure 9a shows that a sequential execution of three kernels on a single processor P1 will consume 53308 cycles, which is the sum of the ker-

nel cycle counts as indicated in italic for each kernel. These cycle counts were obtained by profiling the sequential application. Figure 9b shows a functional pipelining parallelization over three processors, while Figure 9c shows a combined functional pipelining and data-parallel parallelization, in which kernel ME2 is split in two parallel threads. The latter parallelization is more balanced, and hence the combined cycle count is lower, now totaling 18522 cycles. Finally, Figure 9d shows the optimal parallelization over the six processors of our target platform.

Note that it is the developer that explores different parallelizations by indicating how each kernel should be parallelized and assigned to (a) processor(s). Based on the developer's input, our tool can immediately present the estimated total cycle count, and it generates the multi-threaded code which includes all communication code that is necessary to pass data from the L2 memory to each processor's L1 memory and back over the platform's NoC. The resulting parallel program is profiled again, and the tool reports the number of cycles required to perform the required block transfers over the NOC, as shown in Figure 9e. Looking for example at the ME1 kernel, we observe more about a factor 3 difference between maximum cycles required for communication (2154+330) versus computation (9261), indicating that the NoC can be clocked at three times lower frequency than the processors. In addition, different bandwidth can be reserved for the different connections. For example, the ME2 connection requires more than a factor 4 less bandwidth than ME1. By doing this exercise for different parallelizations, the developer can efficiently explore the parallelization design space.

Finally, the resulting code can be executed on our virtual platform model. Figure 10 shows how much of the time each processor spends on different parts of the processing, the communication, and the synchronization. Preamble/postamble are the times some stages of the functional pipeline are idle because the pipelining is being set-up/emptied. Thread synchronization times originates from load imbalances as seen in Figure 9. L1 conflicts stall cycles result from access conflicts of each TI's two load/store units that share a single L1 memory. The block transfer synchronization overhead is very limited, as the corresponding DMA transfers are executed in parallel (prefetching data).

4.2. A Platform for MPEG-4-like Applications

In a second experiment we designed and implemented a programmable multi-mode multimedia platform on which multiple video coding and decoding standards can be implemented such as AVC H.264, MPEG-4, MPEG-2, etc. This platform had to meet the following requirements:

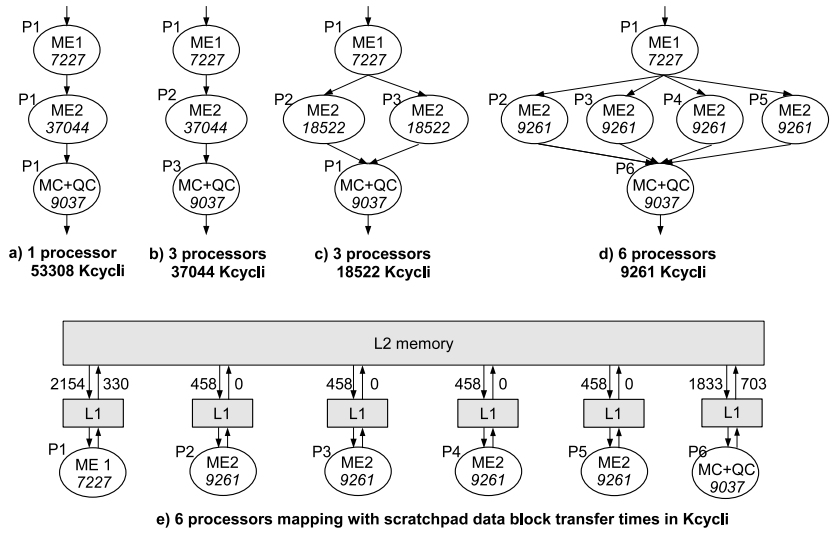

Figure 9. Different QSDPCM parallelizations. Cycle counts are for VGA resolution.

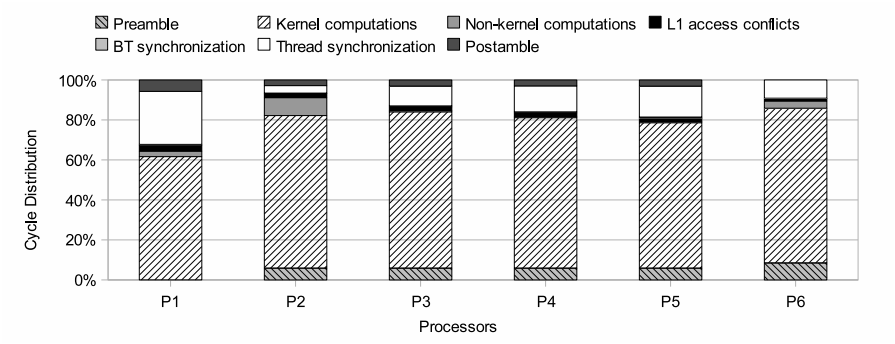

Figure 10. Cycle usage for final SQDPCM implementation at VGA resolution.

Real-time performance Play Baseline Profile level 3.1 AVC HDTV 720p encoding at 30 FPS as well as AVC CIF encoding at 30 fps in combination with up to 4 AVC CIF decoding streams for a 4-user video conferencing).

Composability Run different combinations of applications.

Power-efficiency Components not used at some point are disabled.

Design flow support The platform must support the design flow proposed in this paper by means of a multi-processor architecture with hierarchical shared memories and a scalable interconnect with guarantees for bandwidth of various data transfer streams.

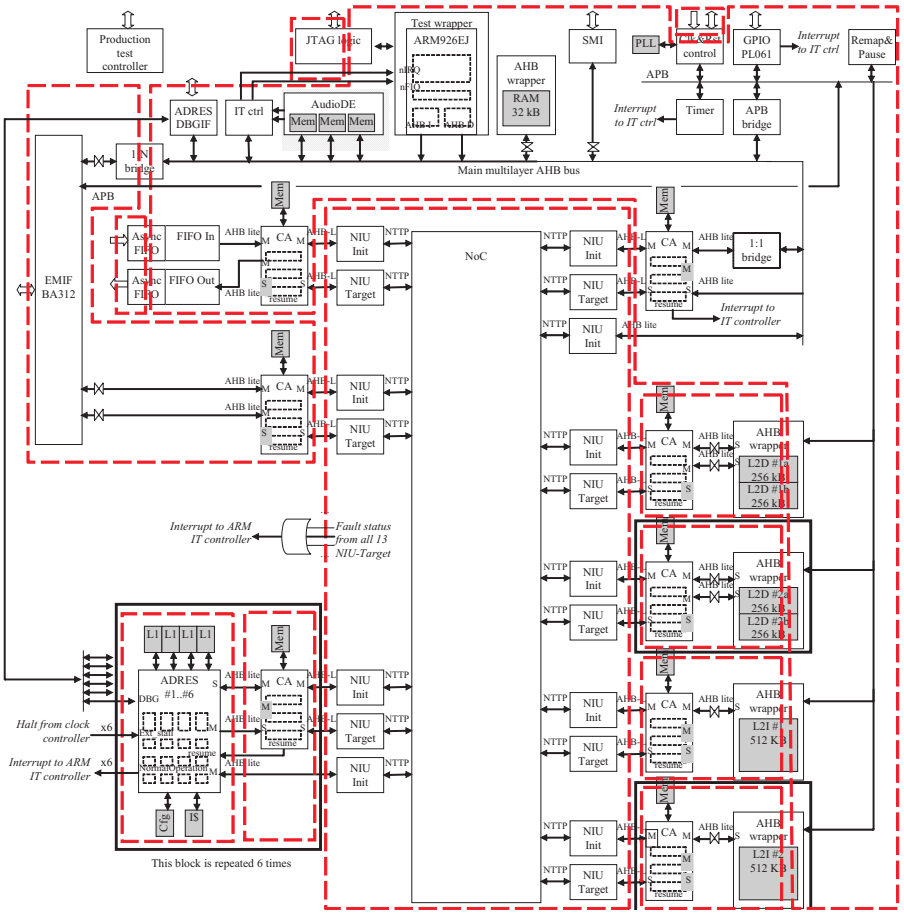

Figure 11. Our multimedia platform is comprised of an ARM subsystem, a FIFO interface, an external memory interface, four Level 2 memories and 6 ADRES cores.

Area The die area should not exceed 100mm^2 in a 90nm process.

Figure 11 provides a block diagram of our platform. Block arrows represent interfaces with the external world. The figure shows the following subsystems which are connected through a central Arteris NoC (http://www.arteris.com). The ARM subsystem performs general control tasks, including system configuration (clock frequencies, voltage island management, clock activation/de-activation), system booting, general data movement (from FIFO interface to and from L3 and between L3 and L2 memories), task scheduling on the various processors in the system, synchronization and communication with the external world, control and relay of debug information from the ADRES processors, and, optionally, audio encoding. The L3 EMIF (external mem. interface) accesses external SDRAM. The FIFO interface is a dedi-

cated communication channel to the external world. Typically raw and compressed data will be exchanged through this channel. Six ADRES core form the intensive processing elements, targeted at computation tasks. Two L2 data memories: these are used to store data at a L2 stage, which are shared by the different actors of the system. Each sub-system is connected to the NoC through a CA. These modules are intended to perform complex 2D data movements autonomously, which they thus offload from the processing elements. These CAs also feature a bypass capability allowing the processing element to perform simple data load/stores through the NoC.

Our platform is divided into several clock domains, which are highlighted by red dotted areas (notice that there is one independent clock per ADRES core). These clock domains have been chosen to enable the adaptation of the platform's resources to the different needs of different applications. For example, some applications are more CPU bound while others are more memory bound, and some applications will allow more parallel prefetching than others. This means that different applications have different optimal ratios between memory operating frequencies, bus (NoC) operating frequencies and processor operating frequencies. The analysis in Section 4.1 revealed one such operating point for the QSDPCM standard. Furthermore, the platform features eight separately powered voltage islands highlighted by a bold border: the six ADRES cores and the L2D2 and L2I2 memory nodes. These voltage islands are not foreseen to be turned on or off frequently (in a dynamic way) as their reaction time is relatively long (voltage ramp-up time, global clock slow down, reset de-assertion phase). Instead the goal is to turn them off (or to lower the frequency of clock domains) for the entire run of an application that does not require all resources. For example, not all six ADRES cores are needed when a video conference with less than six users is set up, or when a single stream of lower resolution than HDTV 720p is being encoded. As such, clock domains and voltage islands ensure efficient scalability in the platform: only the computing performance needed at some point is consuming energy.

4.2.1. *Platform Implementation*

Two goals have been driving our implementation effort. The first goal was a functional verification. To this end, we mapped the platform on an emulator which then ran a compiled MPEG-4. The second goal was an estimation of area, reachable clock frequency and power consumption. We achieved that goal by synthesizing the platform on a technology library. To realize these two goals, a hardware model of the complete platform was required. The NoC (Arteris), the ARM core and the EMIF

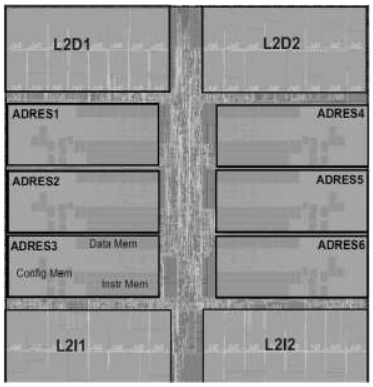

Figure 12. The platform layout: the die size is 64mm^2.

(Barco-Silex) are off-the-shelf components. The other blocks have been designed internally in VHDL.

4.2.1.1. *Emulator mapping* The platform has been mapped onto an high-performance hardware emulation system, the Mentor VStation Pro. The design uses 56% of the available capacity and can operate between 430kHz and 493kHz. The application was tested either by using the ARM AXD debugger, enabling a direct connection to the ARM core on the emulator, or through the FIFO interface for faster execution (the ARM debugger communication requires interactions through the slow JTAG port). Several hundred QCIF frames were encoded to validate the correctness of the platform and of the compiled software mapping.

4.2.1.2. *Synthesis and Place & Route* The platform has gone through a complete ASIC design flow. The targets set forward for the different clocks were: 300MHz for the ADRES cores, 75MHz for the ARM subsystem and 150MHz for other components (CAs, NoC, L2 memories, EMIF, FIFO). The design was successfully laid out using Synopsys Design Compiler and Cadence SoC Encounter, fitting in a die of 8mm x 8mm (Figure 12) and meeting the frequency constraints. The technology library used is TSMC 90nm general purpose process with normal VT. The complete design (including memories) represents 5M gates equivalent plus 2.35 MBytes of L2 and L1 memories.

4.2.2. *Application Mapping*
We mapped an MPEG-4 encoder onto this platform to demonstrate that (1) a complex platform can be designed that operates correctly as predicted, even though some components are suboptimal, and (2) that an application can be mapped onto the platform, albeit suboptimally.

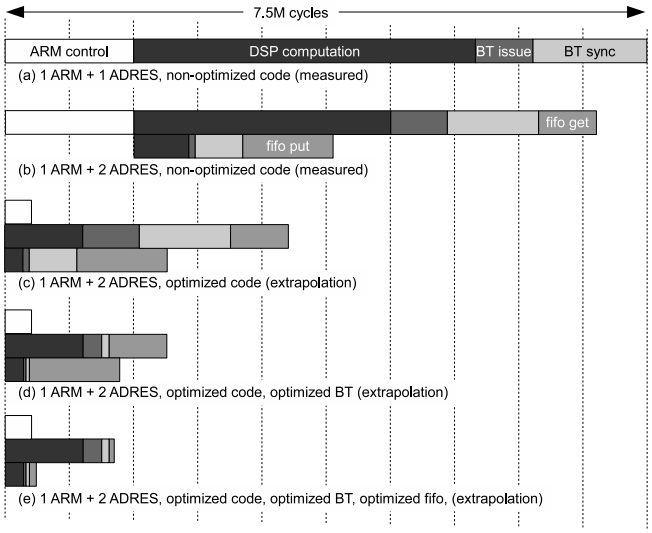

Figure 13. Performance results (number of cycles for one QCIF frame) and extrapolation for the mapping of an MPEG-4 encoder on the platform.

Figure 13 shows some results of this mapping experiment. The top bar shows the execution time breakdown of the controller part of the encoder mapped onto the ARM processor (being the rate control and the fetching of the data to be encoded) and of the computation intensive parts, being the *unoptimized* kernels, mapped onto a single ADRES core. These breakdowns were obtained using virtual platform simulations. Because of the memory hierarchy consisting of L1 scratch-pad memories and a shared L2 memory, even the single-threaded computational part already needs to perform block transfers, which consume about 27% of the time spent on the ADRES core. Because the ARM code in this mapped application fetches the data to be decoded next, i.e. without pipelining or prefetching of data from additional frames, it cannot be executed in parallel with the code running on an ADRES.

The second bar shows the same breakdown for the same application mapped onto one ARM and two ADRES cores. In this version, the motion estimation is run in a separate thread from the other kernels. As such, both threads can run in parallel perfectly. As can be seen from the result, this parallelization is not balanced at all: the motion estimation running on one processor performs about 1/5 of the DSP workload, while the other kernels running on the other processor perform about 4/5 of the DSP work. What can also be observed is that in this experiment, the FIFO communication results in significant overhead. Because of that, the total speedup of the application is limited to only 8%.

While the simulations on the virtual platform and the validation on the VHDL emulator show that the platform performs as expected, the results are clearly all but optimal, reaching only a 8% speed-up after parallelization over 3 threads. Due to the limited amount of resources available at our research institute, we could not optimize the whole platform to get more optimal, measurable results. But to get some idea of where a better mapping and better platform could take us, bars (c) to (e) show extrapolated results based on additional experiments that we performed on isolated parts of the platform.

First, the ARM code can be optimized easily. Instead of fetching one byte of frame data at a time with the *fread* C-library function, buffered fetches can be used. This reduces the workload significantly. Furthermore, by fetching frames one frame ahead of the actual decoder implemented on the ADRES cores, the ARM code can be executed in parallel with the ADRES code. On isolated ADRES code kernels, we have performed additional ILP and DLP optimizations, that reduce their cycle count by an average factor of 3.5. Combining these optimizations with the ARM code optimizations, the result of bar (c) can be obtained.

Next, the CAs can be optimized. Each CA needs to be configured from within the application code using an API to pass the parameters defining the block-transfer to the CA (BT_issue) and its completion is monitored through a second API (BT_sync). The actual block-transfers are executed by the CA (configuration of the remote CA, setting-up DMA channels, monitoring progress and completion of transfer, ...) through firmware executing on the CA's internal controller (in our case, a stripped-down Leon processor). The implementation of the CA (both hardware, firmware and APIs) contains quite some overhead that we are currently removing: for example, we are limiting the parameters that need to be passed back-and-forth between the CA and the application processor (ADRES), simplifying the protocol to configure the remote CA, etc. With these optimizations we expect to reduce the time required for the application processor to execute BT_issue and BT_sync functions by roughly a factor of 6, resulting in the situation shown in bar (d).

Finally, the RT-lib can be reimplemented more efficiently. The parallelization using 2 ADRES processors relies on FIFO communication. As there is no direct hardware support for FIFO communication in the CA, the FIFO behavior is implemented entirely in a run-time library executing on the communicating ADRES processors, leading to a significant processor load. This situation can be corrected by off-loading part of the FIFO communication implementation to the CA resulting in the situation shown in bar (e).

With these three optimizations, we believe we can reduce the required number of cycles from 7.5M per frame to 1.2M per frame. Still further optimization can be obtained of course, by performing a better balanced thread-level parallelization, over more ADRES cores.

4.2.3. *Power Estimation*
Not having a fully optimized application that exploits all platform resources, we present power numbers for the individual components. A layout of a synthesized ADRES core and simulation of an optimally mapped AVC decoder showed that one ADRES core clocked at 300MHz consumes 91 mW of which about 5mW is static power. For a power breakdown of an older, less optimized version of the ADRES core, we refer to (Mei et al., 2008). The ARM data sheet provides a power number of 0.14mW/MHz. Since our ARM is clocked at 75MHz, this implies a power estimate of 10.5mW. For the memories, data sheets provide numbers per read and write access. With an estimate of the numbers of reads and writes required to execute a video codec optimally, we estimate the L2 memories to consume around 40mW (data) and 30mW (instructions). With respect to the NoC, separate experiments have been conducted together with Arteris (Milojevic et al., 2007). The results of that experiment, again for an estimate of the NoC use in an optimally mapped coded, is a power consumption estimate of less than 25mW. Finally, we measured the power consumption of a custom 150MHz CA design (not based on the aforementioned Leon processor, and developed outside our platform by another group in our department) to be 1mW. Combined, this brings us to $6 * 91 + 10.5 + 40 + 30 + 25 + 13 * 1 = 664mW$. This is only an estimate however, as the additional cost of the interconnect is not taken into account.

4.2.4. *Related Platforms*
The SiliconHive HiveFlex VSP2500 platform (VSP, 2007) is a programmable MPSoC platform. The main difference with our platform, apart from the processing cores, is that it features special purpose connections between processors and a shared system bus with DMA controllers. The Nexperia PNX1700 media processor from NXP (PNX, 2005) features a programmable TriMedia TM5250 VLIW CPU and several dedicated accelerators. It features a shared bus to main memory and the VLIW includes a L1 data cache. Texas Instruments DaVinci video processors (Davinci, 2007) include a single programmable TI C64+ VLIW DSP and dedicated hardware accelerators to perform video coding, and an ARM9 processor to run application control. These processors feature a shared memory that is connected to all processors via a so-called switched central resource. ST's Nomadik multimedia

processors, such as the STn8815P14 (nomadic, 2008), feature an ARM9 controller and several (up to 4 at the time of writing) application-specific, programmable DSPs for tasks such as accelerating video processing, audio processing, graphics acceleration, etc. As a communication mechanism they use a multi-layered AMBA crossbar.

We not necessarily want to differentiate purely on a platform basis. We focus instead on the combination of tool flow and platform, for example by using scratch-pad memories for which we have tool support. Our tool support might be applicable to the above platforms as well in so far as they have scratch-pad memories. Of course, parallelization is one thing. Doing it good, in a predictable, composable and scalable manner is another. Compared to the above platform, we believe our NoC-based platform to score much better on these criteria.

5. Conclusions

This paper presents our experience with designing MPSoC multimedia platforms and with the mapping of applications on those platforms. We have discussed the components (DSP processors, scratch-pad memory hierarchies, communication assists and NoCs) that we believe should be used to build such platforms. These components are chosen because they result, when abstracted by a run-time manager, in predictable, composable, scalable platforms that can deliver high performance at high power-efficiency. Based on run-time abstractions like FIFOs and block transfers from and to a shared memory, we have also proposed a design-time tool flow for mapping applications. This tool flow enables the exploitation of all forms of parallelism in a predictable, composable and scalable manner. The feasibility of both the design flow and the proposed platform has been confirmed experimentally for a QSDPCM application mapped on a platform consisting of off-the-shelf components, and for an in-house design of a multimedia platform with state-of-the-art components to support AVC and MPEG-4 video codecs.

References

Aouad, M. I. and O. Zendra: 2007, 'A survey of scratchpad memory management techniques for low-power and -energy'. In: *Proc. of ICOOOLPS*. pp. 31–38.

Baccelli, F., G. Cohen, G. J. Olsder, and J.-P. Quadrat: 1992, *Synchronization and Linearity: An Algebra for Discrete Event Systems*. Wiley.

Baert, T., E. De Greef, E. Brockmeyer, P. Avasare, G. Vanmeerbeeck, and J.-Y. Mignolet: 2008, 'An Automatic Scratch Pad Memory Management Tool and

MPEG-4 Encoder Case Study'. In: *Proc. of Design Automation Conference (DAC 2008)*. To appear.

Benini, L. and G. De Micheli: 2006, *Networks on Chip: Technology and tools*. Morgan Kaufmann.

Betz, V., J. Rose, and A. Marguardt: 1999, *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers.

Burger, D., S. Keckler, and K. e. a. McKinley: 2004, 'Scaling to the End of Silicon with EDGE Architectures'. *IEEE Computer* **37**(7), 44–55.

Burns, S. M., H. Hulgaard, T. Amon, and G. Borriello: 1995, 'An Algorithm for Exact Bounds on the Time Separation of Events in Concurrent Systems'. *IEEE Transactions on Computers* **44**(11), 1306–1317.

Cervero, T.: 2007, 'Analysis, implementation and architectural exploration of the H.264/AVC decoder onto a reconfigurable architecture'. Master's thesis, Universidad de Los Palmas de Gran Canaria.

Chandra, R., R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald: 2001, *Parallel Programming in OpenMP*. Morgan Kauffman.

Chen, T.-C.: 2006, 'Analysis and Architecture Design of an HDTV720p 30 Frames/s H.264/AVC Encoder'. *IEEE Tr. On Circuits and Systems for Video Technology* **16**(6), 673–688.

Cockx, J., K. Denolf, B. Vanhoof, and R. Stahl: 2007, 'SPRINT: a tool to generate concurrent transaction-level models from sequential code'. *EURASIP J. Appl. Signal Process.* **2007**(1), 213–213.

Dasygenis, M., E. Brockmeyer, B. Durinck, F. Catthoor, D. Soudris, and A. Thanailakis: 2006, 'A combined DMA and application-specific prefetching approach for tackling the memory latency bottleneck.'. *IEEE Trans. VLSI Syst.* **14**(3), 279–291.

Davinci: 2007, 'DaVinci Technology Overview'. http::://www.ti.com. Product Brief.

Denolf, K., M. Bekooij, J. Cockx, D. Verkest, and H. Corporaal: 2007, 'Exploiting the Expressiveness of Cyclo-Static Dataflow to Model Multimedia Implementations'. *EURASIP Journal on Advances in Signal Processing* **2007**, Article ID 84078.

Feautrier, P.: 1991, 'Dataflow Analysis of Array and Scalar References'. *International Journal of Parallel Programming* **20**(1), 23–53.

Franke, B., M. O'Boyle, J. Thomson, and G. Fursin: 2005, 'Probabilistic source-level optimisation of embedded programs'. In: *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*. pp. 78–86.

Goossens, K., J. Dielissen, and A. Radulescu: 2005, 'The aethereal network on chip: concepts, architectures, and implementations'. *IEEE Design and Test of Computers* **22**(5).

Goossens, K., J. van Meerbergen, A. Peeters, and P. Wielage: 2002, 'Networks on silicon: combining best-effort and guaranteed services'. In: *Proceedings Design, Automation, and Test in Europe Conference and Exhibition (DATE)*.

Halfhill, T.: 2003, 'Silicon Hive Breaks Out'. *Microprocessor Report*.

Huang, Y.-W.: 2005, 'A 1.3TOPS H.264/AVC Single-Chip Encoder for HDTV Applications'. In: *IEEE International Solid-State Circuits Conference Digest of Technical Papers*, Vol. 1. pp. 128–129.

Kandemir, M. and A. Choudhary: 2002, 'Compiler-directed scratch pad memory hierarchy design and management.'. In: *Proc. of DAC*. pp. 628–633.

Lam, M. S.: 1988, 'Software pipelining: an effecive scheduling technique for VLIW machines'. In: *Proc. PLDI*. pp. 318–327.

Mahlke, S., D. Lin, W. Chen, R. Hank, and R. Bringmann: 1992, 'Effective compiler support for predicated execution using the hyperblock'. In: *MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture*. pp. 45–54.

Marescaux, T., E. Brockmeyer, and H. Corporaal: 2007, 'The Impact of Higher Communication Layers on NoC Supported MP-SoCs.'. In: *First International Symposium on Networks-on-Chips, NOCS*. pp. 107–116.

Mei, B., A. Kanstein, B. De Sutter, T. Vander Aa, S. Dupont, and M. Wouters: 2008, 'Implementation of a Coarse-Grained Reconfigurable Media Processor for AVC Decoder'. *Journal of Signal Processing Systems*. To appear.

Mei, B., S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins: 2003a, 'ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix'. In: *Proc. of Field-Programmable Logic and Applications*. pp. 61–70.

Mei, B., S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins: 2003b, 'Exploiting Loop-Level Parallelism for Coarse-Grained Reconfigurable Architecture Using Modulo Scheduling'. *IEE Proceedings: Computer and Digital Techniques* **150**(5).

Milojevic, D., L. Montperrus, and D. Verkest: 2007, 'Power dissipation of the network-on-chip in a system-on-chip for MPEG-4 video encodin'. In: *Proc. of the IEEEE Asian Solid-State Circuits Conference (ASSCC)*. pp. 392–395.

Muchnick, S.: 1997, *Advanced Compiler Design and Implementations*. San Francisco, CA: Morgan Kauffman Publishers.

nomadic: 2008, 'STn8815P14 Mobile multimedia application processor, data brief'. http://st.com.

Pinto, C.: 2006, 'HiveFlex-Video VSP1: Video Signal Processing Architecture for Video Coding and Post-Processing'. In: *Proc. Eighth IEEE International Symposium on Multimedia*. pp. 493–500.

Plevyak, J., A. A. Chien, and V. Karamcheti: 1993, 'Analysis of Dynamic Structures for Efficient Parallel Execution'. In: *Languages and Compilers for Parallel Computing*. pp. 37–56.

PNX: 2005, 'Nexperia PNX1700 Connected media processor, product brief'. http://www.nxp.com. databrief.

Pugh, W. and D. Wonnacott: 1998, 'Constraint-Based Array Dependence Analysis'. *ACM Transactions on Programming Languages and Systems* **20**(3), 635–678.

Rau, B. R.: 1995, 'Iterative Modulo Scheduling'. Technical report, Hewlett-Packard Lab: HPL-94-115.

Richardson, I.: 2003, *H.264 and MPEG-4 Video compression*. Wiley.

Udayakumaran, S. and R. Barua: 2006, 'An integrated scratchpad allocator for affine and non-affine code'. In: *Proc. DATE*. pp. 925–930.

Udayakumaran, S., A. Dominguez, and R. Barua: 2006, 'Dynamic allocation for scratch-pad memory usin compile-time decisions'. *ACM Trans. on Embed. Comp. Syst.* **5**(2), 472–511.

Verma, M., *Advanced Memory Optimization Techniques for Low-Power Embedded Processors*.

VSP: 2007, 'HiveFlex VSP2500 Series Video Signal Processor for video coding'. http://www.siliconhive.com. databrief.

Wehmeyer, L. and P. Marwedel: 2006, *Fast, Efficient, and Predictable Memory Accesses*. Springer.

Wolf, M. E. and M. S. Lam: 1991, 'A Data Locality Optimizing Algorithm'. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Toronto, ON, Canada, pp. 30–44.