Exploring Parallelizations of Applications for MPSoC platforms using MPA

Rogier Baert*†, Erik Brockmeyer*, Sven Wuytack* and Thomas J. Ashby*†
*IMEC, Kapeldreef 75, B-3001 Leuven, Belgium
rogier.baert@imec.be, tom.ashby@imec.be
†Interdisciplinary Institute for BroadBand Technology (IBBT), B-9050 Gent, Belgium

***Abstract*—This paper presents a tool for exploring different parallelization options for an application. It can be used to quickly find a high-quality match between an application and a multi-processor platform architecture. By specifying the parallelization at a high abstraction level, and leaving the actual source code transformations to the tool, a designer can try out many parallelizations in a short time. A parallelization may use either functional or data-level splits, or a combination of both. An accompanying high-level simulator provides rapid feedback about the expected performance of a parallelization, based on platform parameters and profiling data of the sequential application on the target processor. The use of the tool and simulator are demonstrated on an MPEG-4 video encoder application and two different platform architectures.**

I. Introduction

The design of System-on-Chip platforms is increasingly based on utilizing multiple processors. These MPSoC devices have the potential to provide high performance whilst enabling fast time-to-market, maximizing design reuse and providing flexibility through programmability.

However several serious challenges in designing MPSoCs still remain, such as application partitioning and design space exploration [1]. Another concern is the scalability of the platform and application, i.e. how many processors are required, and, as technology evolves, whether re-designs will be needed.

The exploration methodology presented in this paper aims to facilitate the programming of multi-processors by automating the transformations required to partition an application. It targets data-dominated signal processing applications, such as image and video codecs. These kind of applications often have multiple opportunities for parallelization, both on a functional and data level, but finding the optimal solution is not trivial. By avoiding much of the manual work and by providing quick feedback to the designer, this process becomes much more efficient.

Figure 1 shows an overview of the exploration flow. It starts from the C source code of a sequential application. Profiling data is collected by source code instrumentation and execution on a target platform or instruction set simulator (ISS). Based on an initial Parallelization Specification (*ParSpec*), parallel C code is generated by the MPSoC Parallelization Assist (MPA) tool. A fast, high-level simulator (HLsim) generates an execution trace, using the profiling data and a platform specification. The designer can then adapt the *ParSpec* and/or platform specification based on analysis of this execution trace.

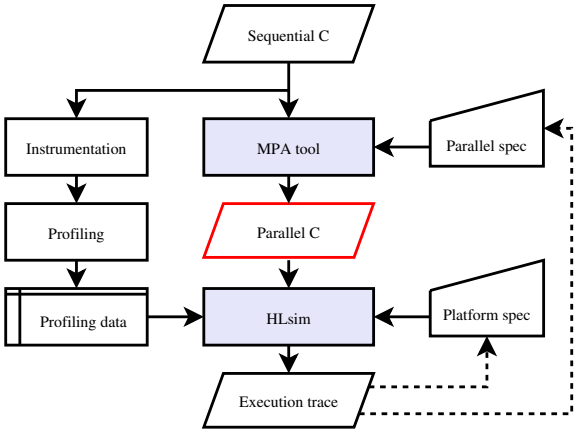

Fig. 1. The exploration flow.

The organisation of this paper is as follows: The MPA tool and generation of traces are introduced in Section II and III, respectively. Section IV details the exploration of different parallelizations of an MPEG-4 encoder using the MPA tool and HLsim. Section V discusses related work. Section VI concludes the paper.

II. The MPA tool

The MPSoC Parallelization Assist (MPA) tool analyzes the application and generates parallel source code based on the directives specified by the designer. The specification, analysis, transformation and execution model are discussed in the following sections.

A. Parallelization specification

The designer specifies the parallelization, which the tool should generate, in a file separate from the application source code. This *ParSpec* refers to labeled statements or blocks in the source code which mark distinct functionalities of the application. By keeping the parallelization specification in a separate file, instead of directly annotating it in the source code, many different parallelizations can be explored and retained, starting from the same source code.

E. Brockmeyer and S. Wuytack are currently with Target Compiler Technologies, Leuven, Belgium

978-3-9810801-5-5/DATE09 © 2009 EDAA

```
parsection ParFrameProc:

  parsecblock label frameProc::parallel_section

    thread me_left:
      looprange iterator v from 0 to 9
      include label frameProc::MotionEstimation

    thread me_right:
      looprange iterator v from 9 to 18
      include label frameProc::MotionEstimation

    thread mc_tc_tu_ec:
      include label frameProc::MotionCompensate
      include label frameProc::TextureCoding
      include label frameProc::TextureUpdate
      include label frameProc::EntropyCoding
```

Fig. 2. An example *ParSpec*.

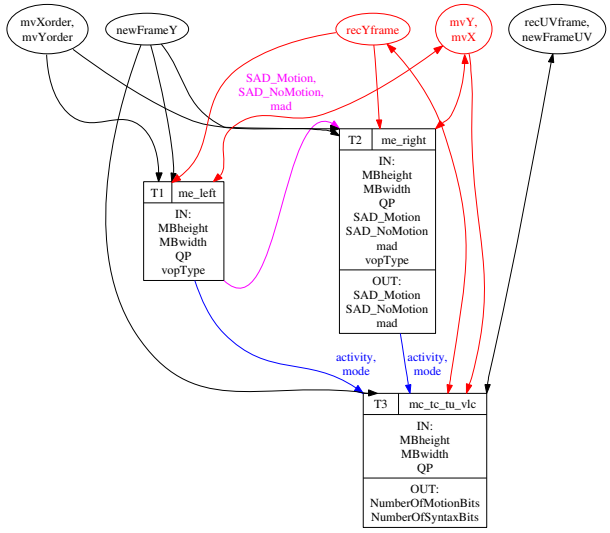

Fig. 3. The dependency graph generated by the MPA tool for the *ParSpec* of Figure 2. The square boxes represent threads, circles represent shared variables. Blue arrows indicate FIFO communication. Red arrows indicate loop-carried dependencies. Purple arrows indicate the communicated variables are recognised as part of a reduction chain. The IN and OUT fields inside the thread boxes are the thread arguments and return values.

The *ParSpec* consists of one or more parallel sections; outside a parallel section the application is executed sequentially by a master thread. All code inside a parallel section must be assigned to at least one thread.

A parallel section can contain both functional and data-level splits. In the first case a functionality is assigned as a whole to a thread. In the latter case a functionality is assigned to a thread for a certain set of iterations of an enclosing loop only.

Figure 2 shows a simple example of a *ParSpec*. It shows a possible parallelization of a typical hybrid video encoder application. It consists of 5 basic functionalities which are distributed over 3 threads. One of the functionalities, the *MotionEstimation*, is split over two threads by assigning the iterations which process the left part of the video frame to one thread and the right part to an other. The other functionalities are grouped together and assigned to the third thread.

In addition to splitting the iterator range into smaller sets of consecutive iterations, it can also be split into interleaved sets by specifying a step size. Also, if the designer wishes to do so, the dependency analysis of certain variables can be circumvented by declaring the variable to be "shared" and manual synchronization added in the form of *LoopSyncs*. This will be discussed in more detail in Section II-D.

B. Analysis

Scalar dataflow analysis is done on the sequential application, for the function(s) containing the parallel sections and all functions (indirectly) called from the parallel sections. This results in Factored Use-Def (FUD) chains [2] for all accessed variables. From these FUD chains, the flow (Read after Write), anti (Write after Read), and output (Write after Write) dependencies can easily be derived.

By default each thread gets a local copy of the variables it accesses. These copies can be stored in the local memory of the processor executing the thread, resulting in efficient memory accesses without interference from the other threads. This also means that the kernel profiling results obtained from the single threaded simulation can be reused for the multi-threaded case.

Each time a flow dependency crosses a thread boundary, the last definition has to be communicated from the producing thread to the consuming thread. This is done by inserting communication primitives into the generated partitioned code. We use FIFO style communications channels between threads in a parallel section. Flow dependencies crossing the parallel section boundary (i.e., between the master thread and one of the slave threads) result in input and/or outputs of the slave threads which are communicated via thread arguments and return values when spawning and joining.

For the majority of the variables, each thread accessing it has a local copy and therefore anti dependencies are not relevant. Only for shared variables, i.e., those not handled by the tool, do they have to be taken into account and the necessary synchronization (not communication) has to be added (i.e., a *LoopSync*). Also pure output dependencies, i.e. data that is unconditionally and completely overwritten at the destination, can be ignored for non shared data. Conditional and/or partial output dependencies are treated in the same way as flow dependencies, i.e., the last produced value is communicated via a FIFO.

Statements involving variables with thread boundary-crossing dependencies are represented as nodes in a parallelization model. For all dependencies that potentially need communication (or synchronization), all nodes executing the source of the dependency, and all nodes executing the destination of the dependency are computed. For each node the threads executing it and the corresponding iteration domains are derived from the parallelization model. For each destination node, all source nodes that produce its value are collected. If one of the source nodes is executed by the same thread as the destination node, communication happens inside the thread and no communication/synchronization has to be added.

Otherwise, the iteration domains of the source nodes have to be compared with the one of the destination node to determine overlap. If there is an overlap, a FIFO is created for the variable to be communicated between the two threads, right after the kernel containing the source node. If there is no overlap, no communication is needed.

Besides the source code, the tool also generates a graphical view of the parallelization which shows the dependencies it has found. An example of such graph is shown in Figure 3.

C. Transformation

After the analysis, all required acts of communication are known. All variables to be communicated between the same two threads at the same point in the code are combined into a token and will be communicated via the same FIFO, to minimize the number of FIFOs and as a heuristic to help reduce the per-token overhead. If the iteration domains of the threads at the source and the destination of the FIFO are not the same, guard conditions based on the iterator values are added around the put and/or get function calls to make sure that all tokens put in the FIFOs will also be consumed and vice versa. The FIFO depths are determined based on the parallelization and the *LoopSyncs*. This is a different problem from classical buffer allocation problems for flow graph models, such as Synchronous Data Flow [3], because the form of the flow graphs is restricted as a result of them being derived from sequential programs, and because of the presence of LoopSyncs.

For each thread, a thread function is created containing all the code that it has to execute, the communication and synchronization, and the input and output variables. The boundaries of the loops that are data split are adjusted, and the parallel section is replaced with thread spawning/joining code.

Subsequently, reduction chains of associative/commutative operations (e.g., addition, min/max computations) crossing thread boundaries are detected and the computations are reordered to reduce the dependencies between the different threads as much as possible.

Finally, a configuration function is created that configures all the parallel section, thread, FIFO, and *LoopSync* data structures for the parallel application.

The tool avoids transforming kernels identified by the designer (typically inner-loops or loop-nests), so as not to disrupt any back-end compiler optimizations, such as software pipelining.

D. Communication and Synchronization

The main method of communicating data and synchronizing threads is through first-in-first-out (FIFO) buffers. However, if the designer wants to specify her own synchronization for accessing a certain variable, she can declare this variable shared and specify a *LoopSync*. This can be useful when a dependency, often between different threads in a data-level split, is in fact not as strict as it appears to be from the static analysis. A *LoopSync* specifies a skew or a range of allowed skews between one or more loops whose iterations are distributed over two or more threads. It is essentially a pair of dependence vectors denoting the lexicographical distances of flow- and anti-dependences that must be respected for the loop.

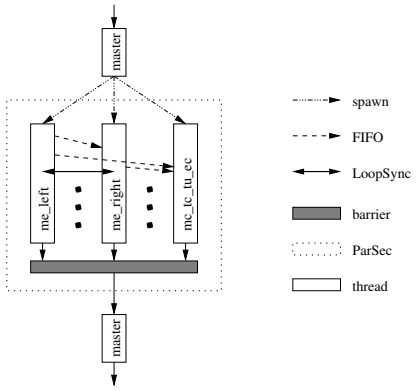

Fig. 4. The execution model. An application can contain multiple parallel sections. When in a parallel section, there can be point-to-point communication between threads.

Often the need for a *LoopSync* occurs because the static analysis only considers arrays as a whole, not each element individually. Future work would include extending the analysis to avoid such false dependencies, or to enable checking the correctness of a manually specified *LoopSync*.

The execution model, depicted in Figure 4, currently assumes that a processing core is assigned a single thread, i.e. all functionalities that are mapped to a core are statically scheduled in one thread. This avoids scheduling overhead and increases predictability.

The MPA tool generates code which targets a run-time library of which the implementation can be optimized for a particular platform. The main functions of this run-time library are thread and FIFO management, *LoopSyncs* and platform configuration.

At the start of a parallel section, all threads are started and FIFO's are initialized, if needed. Just before the end of the parallel section, all threads are synchronized through a barrier and joined again with the master thread. During the execution of a parallel section, the threads operate in an asynchronous fashion, synchronizing only with other threads when needed for communication, either through FIFO channels or *LoopSyncs*.

III. Execution environment

Currently two execution environments for the parallelized applications have been implemented. If the target platform is available, either in the form of real hardware or as a Virtual Platform, this can be used to evaluate a solution generated by MPA.

In case the target platform is not available or simulation is too slow, or if the goal of the exploration is to define the platform, a High-Level Simulator can be used to evaluate a solution.

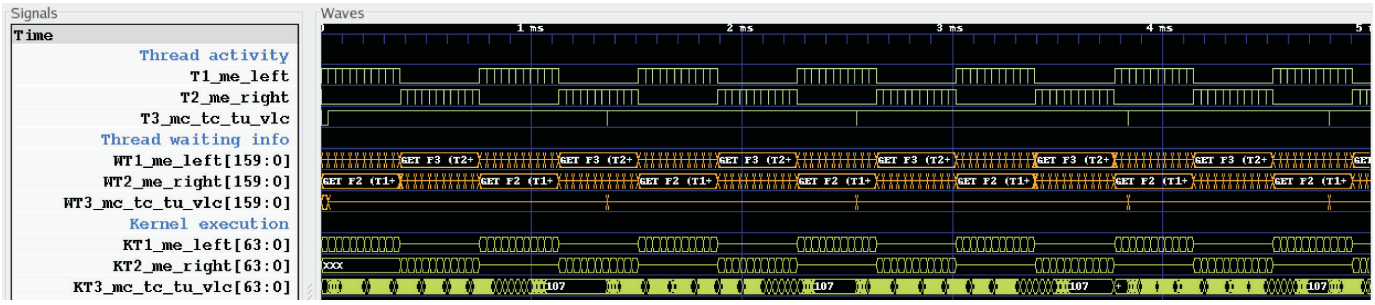

Fig. 5. An example VCD trace generated by HLsim. In this case it reveals that two of the threads in fact never execute concurrently due to a dependency communicated via FIFO's F2 and F3. A solution can be to declare the variable creating this dependency as shared, and protecting access to it with a *LoopSync*.

A. Native platform

In case the target platform is running an operating system (OS), the RTlib can be implemented on top of the native thread functionality provided, such as Pthreads. The scheduling of the threads is thus dynamically handled by the OS. This can create some overhead, but experiments on the Linux OS have shown that this is minimal. If the platform is not running an OS, a fixed thread-to-processor assignment is used.

B. High-level simulation

The high-level simulator (HLsim) enables the designer to quickly evaluate a specific parallelization based on the application profile and some platform parameters.

1) Application profiling: The principle of HLsim is based on annotating the kernel profiling information back into the parallel code by means of function calls. The implementation of these calls can be changed to select recording or playback of the profiling data. Recording is best done using a cycle-accurate Instruction Set Simulator (ISS), extended with a memory-mapped I/O-module or foreign-function interface for interfacing with the profiling library, which executes in host context, to ensure minimal interference with the program execution.

The control flow path is stored along with each sample, such that the correct sample can be found for each call, even if the code has been parallelized. Because the profiling data is sample based, and not averaged, all the dynamic behaviour and correlations between samples are preserved.

2) Platform modelling: In addition to kernel timings and synchronization, delays for data transfers through FIFO's can be estimated based on the size of the data and the platform parameters. However it does not take into account the effects of a cache. Even though the absolute results may therefore be inaccurate, the relative results, such as speed-up, can be used in the exploration of the design space.

The application is executed using a simulation kernel which models timing and concurrency, such as SystemC or Tipsy [4].

3) Simulation results: The simulation will generate a VCD trace file in which thread and kernel activity, FIFO fill levels and *LoopSync* status are traced. The designer can analyze this data to see if any bottlenecks occur and whether the parallelization is well balanced. Based on this analysis either the platform or the parallelization can be adjusted.

Figure 5 shows a part of such a trace file. The signal representing a thread is high when it is active. The "thread waiting info" signals indicate why a thread is inactive, for example because it is blocked when trying to read from an empty FIFO, write to a full FIFO, or is outside its allowed iteration range of a *LoopSync*.

In addition to the VCD trace, a report is generated that summarizes the overall results, and statistics on thread and FIFO activity, etc. The simulation speed for our example is about 1 QCIF frame per second for a 6-thread MPEG-4 encoder application on a 3.2GHz Pentium 4 PC.

IV. MPEG-4 Encoder Case Study

To demonstrate the capabilities of the MPA tool, an exploration of different parallelization options for a video encoder was undertaken. First, we demonstrate the feasibility of the execution model on an actual multi-core platform. Secondly, we show the correlation between the HLsim estimations and the actual results for the same platform. Finally, an exploration is done for an embedded platform.

The application which is used for the experiments is an MPEG-4 SP encoder [5], an industry standard hybrid video encoder.

A. Feasibility of the execution model

Although the methodology targets embedded platforms, current cache-coherent shared memory MPSoCs only have a limited number of cores. Therefore an 8-core x86-based platform, consisting of two 2.1GHz quad-core AMD "Opteron" processors running the Linux operating system, is used to show the feasibility and scalability of the execution model.

A set of parallelizations is generated ranging from one to seven threads. Using the Pthreads based RTlib, the execution times are measured for a sequence with 4CIF resolution. Figure 6 shows the measured speed-up relative to the single threaded implementation. For two and three threads, the processor effective utilization is between 70 and 75 percent, for four to six threads it is around 60 percent. The seven thread parallelization does not improve the execution time anymore, which can be attributed to the overhead of data communication between threads and the resulting cache effects.

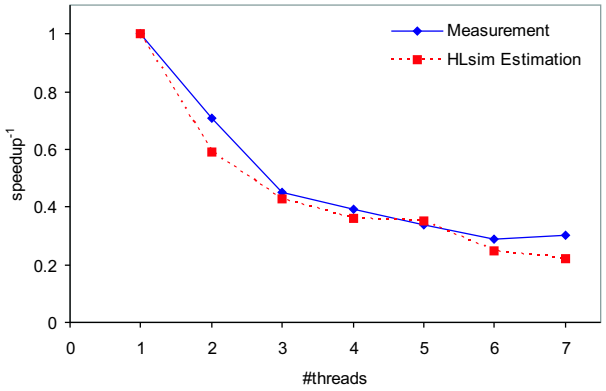

Fig. 6. The measured and estimated speedup of the MPEG-4 encoder application. Along the vertical axis is the relative execution time.

B. HLsim validation

The same set of parallelizations as used for the speed-up measurements is used to estimate the speed-up using the High-Level Simulator. The results, shown in Figure 6, show a strong correspondence between the HLsim estimation and the actual measurement. Only with the seven thread implementation the HLsim incorrectly predicts an improvement in execution time.

C. Exploration

The general approach to traversing the design-space is to first make an initial *ParSpec*, based on the required performance and the top-level profile report. First, functionalities should be data-split if a single processor can not deliver the required performance; secondly, functionalities can be grouped together in order to achieve a balanced distribution of compute cycles. After this first iteration the efficiency can be assessed by inspecting the trace file and generated report.

If a data-split into equally sized, consecutive parts results in unbalanced execution, then probably the execution times are not evenly distributed over the range which is split. This may occur for example in the MPEG-4 encoder when a specific area of the video frame contains more motion and thus the motion estimation requires more processing time. A solution can be to choose the data-split at a different loop level, and/or to make an interleaved split where loop iterations are allocated to threads in a cyclic manner rather than a block. Both types of split are supported by MPA.

Once the application had been prepared, i.e. annotated with kernel labels and profiled, the exploration of the parallelization design space using HLsim can be performed in about half a day.

1) ARM11 RISC: Considering the execution profile of the ARM11, a RISC processor for embedded applications, in Figure 7, it is clear that the MotionEstimation (ME) is dominating the overall performance; it consumes almost 2 times the number of cycles of all other kernels together. So if the ME is split off, and executed in parallel with the rest of the application, a speed-up of 1.4 can be expected (Figure 8B).

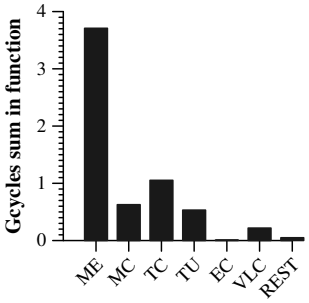

Fig. 7. The function cycle distributions for ARM11. On the horizontal axis the top level functionalities are shown. The vertical axis shows the total contribution of a functionality to the overall cycle count.

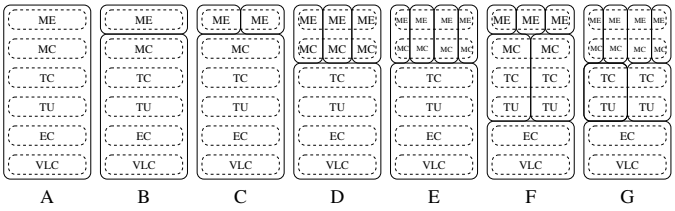

Fig. 8. A schematic overview of the ARM exploration, displaying 7 different parallelizations (A to G). Dashed lines indicate functionalities; solid lines indicate threads. Thus, on the left side (A), the sequential implementation is shown.

The next step is to apply a data-split. This can be easily applied to this ME; but the designer has to take into account that the efficiency of a data-split is often not 100%: pre- and post-amble for pipelined execution, load imbalance and synchronization overhead decrease the efficiency (Figure 8C).

To start with, a moderate data-split into two is explored, which already results in a further speed-up (see the 3-thread point in Figure 9). Further refinement shows that combining the ME and MC functionalities often leads to better results, because these functionalities share much data, such as the video frames, which otherwise has to be communicated (Figure 8D).

Figure 8 shows all "Pareto-optimal" parallelizations as found by the exploration. Many more have been evaluated, but are omitted for brevity. It shows that a wide range of solutions, both in terms of performance and cost, can be generated.

V. Related work

There are many possible techniques for programming multi-processor system-on-chips, but most of them are derived from either scientific computing, a field in which large distributed systems have been commonplace for many years, or from hardware design, which is inherently parallel. This section restricts itself to programming techniques based on the C programming language, since this is the most widely accepted language, especially for programming DSPs.

The closest technique to our methodology from the scientific computing category is OpenMP [6]. It allows a designer to add pragmas to the source code that specify which parts can be executed in parallel. A compiler or pre-processor will translate these pragmas into a multi-threaded application, usually targeting a POSIX threading library. However, OpenMP assumes

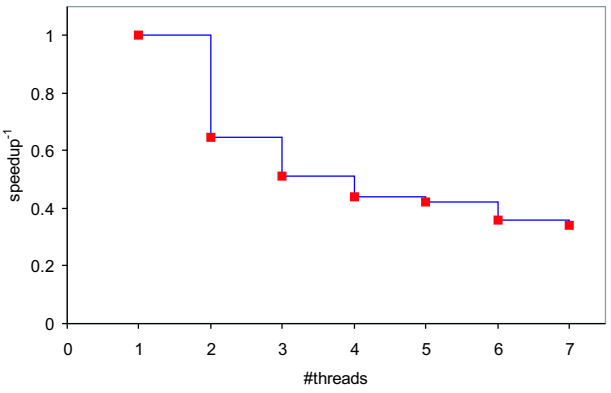

Fig. 9. Pareto curve of the parallelizations corresponding to Figure 8.

that there are no dependencies within parallel sections. In practice this is seldom the case, especially at a coarse task level. It is then up to the designer to either remove these dependencies by, for example, loop-skewing, or to explicitly handle the communication and synchronization of data between threads. Besides the fact that this can be error prone and has to be redone for every parallelization, it also defeats one of the major design goals of OpenMP, namely incremental parallelization of the application (independently of the exact run-time environment) using a high level of abstraction.

A tool which inspired some features of the MPA tool is SPRINT [7]. It uses labels in the source code to let the designer specify a task level functional pipeline. It extracts dependencies by static analysis of the code and inserts FIFO's to handle them correctly. The tool generates a transaction level SystemC model, in which each task can be further refined into a hardware or software implementation. It does not support data level splits. Also the designer has to manually add timing estimates for each task, which makes it less practical for exploring many different parallelization options.

In [8] a compiler is presented which targets multi-core DSP platforms. It automatically parallelizes loop-nests by applying strip-mining. In addition it partitions the data structures for assignment to distributed memories. However this approach can in general not exploit coarse grain parallelism, nor functional parallelism. Only benchmark kernels were reported to be successfully parallelized.

A methodology for exploring parallel mappings of applications is presented in [9]. It concludes that a combination of task and data parallelism gives the best results in terms of power consumption. The mapping is however still a manual process. The MPA tool and design flow presented in this paper could accelerate the methodology.

VI. Conclusions

The contribution of this paper is threefold. First the MPA tool for parallelizing an application on a coarse task level is presented. Parallelization is based on designer specification, and combining both functional and data-level parallelism. Secondly, a high-level simulator is presented to allow a designer to quickly evaluate a parallel mapping generated by the MPA tool. Finally, a case study demonstrates the use and usefulness of the MPA tool and HLsim by applying them to the industrially relevant MPEG-4 encoder application.

Additionally, the MPEG-4 case study shows the execution model targeted by MPA can result in good speed-up of the application. Many solutions can be explored, spanning a wide range of platform configurations (1 to 7 threads), and resulting in a Pareto curve of solutions from which the designer can choose.

Future work on the MPA tool will focus on adding support for distributed shared memory management, which will remove the requirement for coherent caches for shared data by assigning data structures and scheduling transfers at design-time. This will also increase the predictability and thus the accuracy of the HLsim.

Acknowledgements

This research has been carried out in the context of IMEC's nomadic embedded systems programme, which is partly sponsored by Samsung, the MOSART project (European ICT FP7 Computing Systems) and the MNEMEE project (European ICT FP7 Embedded Systems Design).

References

[1] G. Martin, "Overview of the MPSoC design challenge," in *DAC '06: Proceedings of the 43rd annual conference on Design automation*. ACM, 2006, pp. 274–279.

[2] E. Stoltz, M. Gerlek, and M. Wolfe, "Extended SSA with factored use-def chains to support optimization and parallelism," in *Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*, vol. II: Software Technology, 4-7 Jan 1994, pp. 43–52.

[3] S. Stuijk, M. Geilen, and T. Basten, "Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs," in *DAC '06: Proceedings of the 43rd annual conference on Design automation*. New York, NY, USA: ACM, 2006, pp. 899–904.

[4] D. Verkest, J. Kunkel, and F. Schirrmeister, "System level design using C++," in *DATE '00: Proceedings of the conference on Design, automation and test in Europe*. ACM, 2000, pp. 74–83.

[5] ISO/IEC 14496-2, "Coding of audio-visual objects – part 2: Visual," 2001.

[6] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald, *Parallel Programming in OpenMP*. Morgan Kaufmann, 2001.

[7] J. Cockx, K. Denolf, B. Vanhoof, and R. Stahl, "SPRINT: a tool to generate concurrent transaction-level models from sequential code," *EURASIP J. Appl. Signal Process.*, vol. 2007, no. 1, pp. 213–213, 2007.

[8] B. Franke and M. O'Boyle, "A complete compiler approach to auto-parallelizing C programs for multi-DSP systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 3, pp. 234–245, 2005.

[9] K. Danckaert, K. Masselos, F. Catthoor, and H. de Man, "Strategy for power efficient combined task and data parallelism exploration illustrated on a QSDPCM video codec," *Journal of Systems Architecture*, vol. 45, no. 10, pp. 791–808, 1999.