Using the MegaBlock to Partition and Optimize Programs for Embedded Systems at Runtime

João Bispo[1], João M. P. Cardoso[2],

[1] IST/Universidade Técnica de Lisboa, Av. Rovisco Pais 1, 1049-001 Lisboa, Portugal
joaobispo@gmail.com,
[2] Universidade do Porto, Faculdade de Engenharia, Departamento de Engenharia Informática, Rua Dr. Roberto Frias s/n, 4200-465 Porto, Portugal
jmpc@acm.org

Abstract. This paper presents our recent research efforts addressing the dynamic mapping of sections of execution to a coarse-grained reconfigurable array (CGRA) coupled to a General Purpose Processor (GPP). We are considering the common scenario of a GPP – a RISC processor – using the CGRA as a co-processor to speedup applications. We present a partitioning scheme based on large traces of instructions (named Megablock). We show estimations of the speedups achieved by considering the Megablock.

Resumo. Este artigo apresenta os nossos esforços mais recentes em relação ao mapeamento dinâmico de secções de programas a correr em processadores de âmbito geral (GPPs) para agregados reconfiguráveis de grão grosso (CGRAs). Na abordagem actual consideramos um cenário em que temos um GPP – processador RISC – que utiliza um CGRA como co-processador para acelerar aplicações. Apresentamos um método de particionamento baseado em grandes blocos de instruções (denominados MegaBlocos) e mostramos valores estimados de acelerações do tempo de execução quando se considera o MegaBloco como unidade de partição.

Keywords: Reconfigurable Computing, Dynamic Mapping, Just-In-Time Compilation, Binary Translation.

1 Introduction

The execution of applications on general purpose processors (GPPs) can be enhanced – e.g., lower execution time, lower energy consumption – by moving computationally intensive parts (hot-spots) to specialized custom hardware components such as Reconfigurable Processing Units (RPUs) [1, 2]. This is becoming common practice in high-performance embedded systems. It is common to use a programmable processor, often a RISC-like GPP, to run the application and use a custom hardware coprocessor (e.g., CGRAs – Coarse-Grained Reconfigurable Arrays) when certain requirements cannot be met by the GPP alone.

However, to be able to use the custom hardware, we must rewrite part of the application and explicitly call this hardware when needed. This can be accomplished

INForum 2010 - II Simpósio de Informática, Luís S. Barbosa, Miguel P. Correia (eds), 9-10 Setembro, 2010, pp. 699–710

by several means (e.g., manually by a programmer, automatically by a compiler). By using techniques from both binary translation and dynamic compilation, it is possible to translate the application and insert calls to the hardware at runtime. This way we can transparently move critical sections of programs running on a GPP to a CGRA co-processor without pre-changing the program binaries. We refer to this as dynamic mapping. It has already been successfully applied [3], but research efforts about the benefits and the feasibility of dynamically partitioning binaries to reconfigurable architectures are relatively recent [4].

This paper shows our most recent efforts on dynamic mapping. We identify a set of characteristics that when present in the critical sections can benefit dynamic mapping, and we propose a novel partitioning method which can extract blocks of instructions [5] with those characteristics in mind. We show how this partitioning method can impact performance.

This paper is organized as follows. Section 2 introduces the dynamic mapping problem and motivation. In Section 3 we explain our approach to dynamic mapping and we propose the MegaBlock partitioning method. Section 4 presents experimental results regarding our approach and Section 5 introduces related work. Finally, Section 6 concludes the paper.

2 Dynamic Mapping

As previous work has shown, if we move the critical loops of a program to dedicated hardware units, we can have significant performance improvements [6]. There have been many proposals on accelerators for reconfigurable computing, as well as a plethora of architectures [7, 8]. Most well-known examples include Adres [9], Morphosys [10], Chimaera [11], and XPP [12]. Each one of these architectures proposes unique features and tries to address faster execution and/or energy savings for a set of algorithms. Currently, there is a wide choice of hardware accelerators and fine-grained reconfigurable fabrics such as FPGAs (Filed-Programmable Gate Arrays) are a fairly cheap technology to implement them. The main obstacle to custom hardware units is the significant cost of rewriting the programs to take advantage of those units.

A common approach has been to develop tools which automatically partition a program (typically in C) into software and hardware parts [13, 14]. With the help of profiling information, the tools detect small sections of code where the program spends most of its time (critical kernels or hot-spots). This approach is applied at compile time (statically). Since it is static, it can use more complex algorithms than dynamic approaches. On the other hand, the binary generated by the tools is often tied to a very specific setup. Even when the tool supports several families of the same architecture (e.g., with variations in the number of functional units), at compilation time the options usually are compiling to a very specific architecture, or to the lowest common denominator. In addition, if the execution of a program is sensitive to changes in the input data, the information collected during profiling might not hold between executions, limiting the adaptability of the generated binary.

During static partitioning, we can only move parts of an application to hardware if we have access to its code. In this approach, pre-compiled libraries (e.g., DLLs) are usually out of the partitioning scope, and consequently they are not considered for target-specific compiler optimizations.

Dynamic mapping can make RPUs transparent without compromising existent binary portability, expose more optimization opportunities and expand the use of reconfigurable hardware in embedded computing systems. However, dynamic mapping represents a difficult challenge, since it implies we need to execute many of the tasks performed by static partition at runtime. On the other hand, it provides access to information previously not available, which can be used for further optimizations.

We are considering the common scenario of a GPP using a co-processor to speedup applications. In such a case, the execution will be switching back and forth between the GPP and the co-processor.

We focus our work on the level of the instructions executed by the GPP. By working at a higher level (e.g., doing the partitioning of the program on C code) we might not have access to important information about the execution flow of the program.

3 Our Approach to Dynamic Mapping

The main objective of our work is to contribute in bridging the gap between software and reconfigurable hardware. Embedded computing is a good target since it is an area where it is common to find systems including customized hardware modules and reconfigurable hardware.

We want to move parts of programs to hardware to improve one or more particular aspects (e.g., execution time, energy consumption). So, instead of starting with the hardware and propose a specific architecture, in our approach we want to start with the programs, and discover what kind of opportunities they have for dynamic mapping.

Nonetheless, the particular mapping techniques will depend on the target co-processor architecture, memory interface, and available communication. We think that to maximize the impact of dynamic mapping, we should go beyond the Basic Block and be prepared to map blocks of instructions with dozens to hundreds of instructions. Bearing this in mind, we choose to base our work on the general architecture shown in Figure 1. This kind of target architectures with a RISC-like GPP is commonly used in embedded systems. Currently, we use the Xilinx MicroBlaze *softcore* processor [15] as the GPP to run the programs. Dynamic mapping could possibly be applied to other types of hardware co-processors, but we choose to focus our work on CGRAs, since they generally need less mapping efforts than finer-grained alternatives (e.g., FPGAs). Note, however, that this does not constrain the use of FPGAs as CGRAs can be mapped to the FPGA hardware resources, which is a trend in the reconfigurable computing area.

We present in this paper a novel approach to one of the challenges in dynamic mapping: identifying what portions of code should be mapped (partitioning).

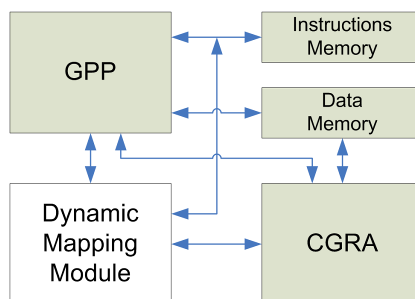

Figure 1. General Architecture

Regarding the architecture assumptions previously described in this section, we identified three issues.

Firstly, the longer a segment of code executes in the co-processor uninterruptedly, the higher the impact of dynamic mapping. This will reduce the communication and reconfiguration overhead [13], and since each partition will incur in a mapping cost the first time it is found and a reconfiguration cost every time it is used, it is desirable to find mapping candidates which will have a large number of iterations.

Secondly, as stated by Amdahl's Law, we need to move a large portion of the program execution to the co-processor if we want to have a significant impact. E.g., for a speedup of 2×, we will always need to move more than 50% of the execution from the GPP to the co-processor.

Lastly, branches are a common occurrence in code running on GPPs, and do not translate well to the usually highly parallel, data-flow co-processors. Branches can also prevent several optimizations and limit the amount of Instruction-Level Parallelism (ILP). Hardware accelerators work best when the control-flow is very low or non-existent.

Taking these issues into account, we consider that a good candidate for mapping would be a segment of branchless code (control-flow issue) which repeats itself a high number of times during execution (iteration issue). It is important that such segments represent a significant portion of the program execution (coverage issue). Figure 2 illustrates the segments we are currently identifying in an execution trace, in an example in pseudo-code.

The BasicBlock is formed by a sequence of instructions with single entry-point and single exit-point – basic blocks end when a branch or jump instruction appears.

A similar, yet more powerful type of segment is the SuperBlock. SuperBlocks are regions of code with single entry-point and multiple exit-points. Originally, it was proposed as a technique to extract more ILP from static compilation [16], but it was later adapted for dynamic compilation [17]. The dynamic version of the SuperBlock represents a common, biased path along several BasicBlocks. A SuperBlock is built by adding BasicBlocks until we reach a BasicBlock that ends with a backward jump. The jump starts a new SuperBlock.

Expanding on the idea of the SuperBlock, we propose another type of segment, the MegaBlock, as a sequence of SuperBlocks, with a bias towards consecutive repetitions. A MegaBlock is built by identifying a sequence of SuperBlocks, up to a

predetermined size. When a sequence of SuperBlocks repeats itself at least one time, that sequence is considered as a MegaBlock with multiple iterations. SuperBlocks which do not form repeatable sequences are also considered as MegaBlocks, albeit with only one iteration. It should be noted that when these three kinds of segments are considered individually, they only have one execution path and the only control-flow inside the blocks are side-exits.

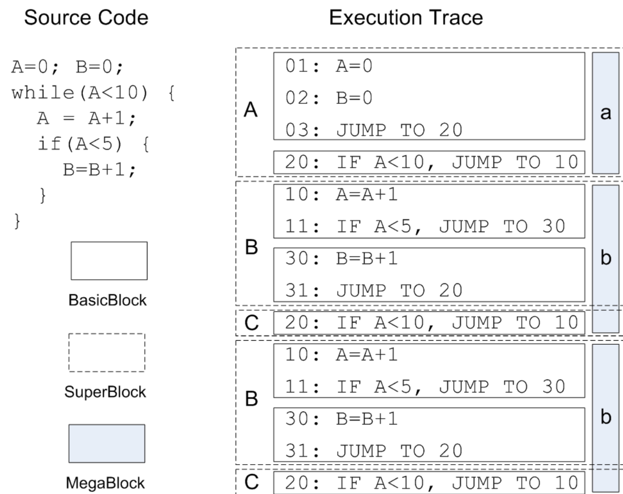

Figure 2. Program execution partitioning according to BasicBlocks, SuperBlocks and MegaBlocks

To use these segments in dynamic mapping, it is important that we can detect and extract them during runtime. To detect BasicBlocks, we identify branch instructions. SuperBlocks can be detected by identifying backward branch instructions. To find MegaBlocks at runtime we propose a technique which first, uniquely indentifies SuperBlocks by using the first addresses of their BasicBlocks to create a hash value (e.g., [16]). Using a hardware pattern matching module, we can efficiently find MegaBlocks within a stream of SuperBlock hashes. We could make the detection of MegaBlocks over the BasicBlocks instead of the SuperBlocks, but the coarser granularity of the SuperBlock reduces the pattern matching requirements significantly.

Before mapping a section of the program execution, we can apply optimizations to expose more ILP, or to reduce the number of instructions to map. Although this is not explored in this paper, we will refer some considerations about these optimizations. Since the algorithms should perform during runtime, we favor algorithms which map well to hardware. We focus on algorithms which can be applied to a stream of instructions and which use tables to store temporary data, (instead of, e.g., graph representations). The reason is that, later they might be easier to translate to hardware. We use a simplified Single-Static Assignment (SSA) format without Phi functions [18], and maintaining the original number of the registers added with the number of each specific definition. It is simpler as the algorithms are applied over blocks of

instructions with a single execution path. Since there is no more than one path at any given moment, a use can only be reached by a single definition – which can be kept in a table – removing the need for a Phi function.

4 Experimental Results

We chose a set of 13 benchmarks which are commonly used in embedded computing and which represent a wide range of integer computations. The benchmarks used are: adpcm coder and decoder, autocorrelation, bubble sort, discrete cosine transform, dot product, fdct, fibonacci, fir, max, pop_cnt, sobel and vecsum. All benchmarks were compiled with *mb-gcc* (the GCC compiler targeting MicroBlaze) using different levels of optimization. The number of instructions executed for the benchmarks range from around 500 instructions to 300,000 instructions.

The Megablock identification uses the maximum size of the sequence of SuperBlocks as a parameter. Figure 3 represents the coverage of MegaBlock based partitioning with different maximum sequence sizes. Table 1 shows the sizes, in number of executed MicroBlaze instructions of the corresponding MegaBlocks.

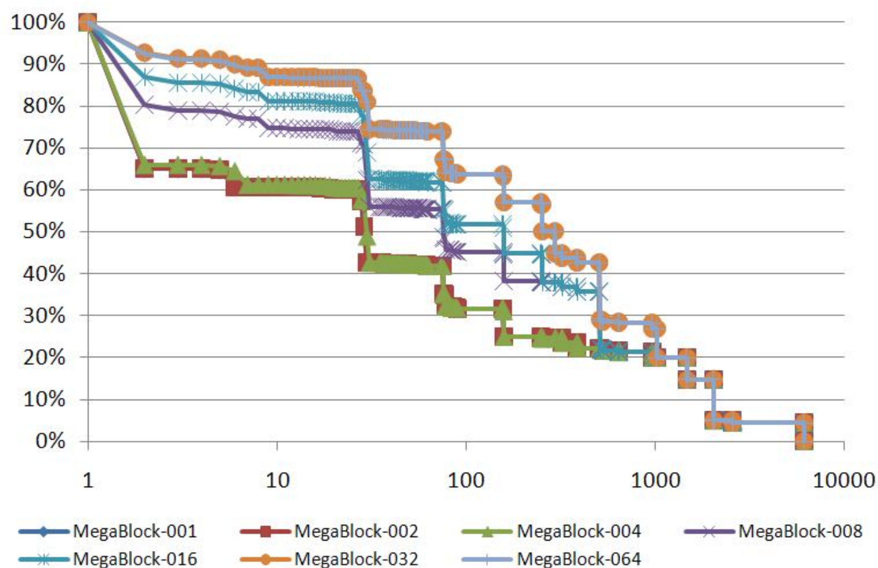

Figure 3. Portions of the program execution (*Y axis*) that are covered by MegaBlocks which have *at least* a certain amount of iterations (*X axis*), according to the maximum number of SuperBlocks a MegaBlock can have. Since every block has, at least, one iteration, value 1 on the *X axis* corresponds to 100% of the program execution on the *Y axis*

Figure 3 and Table 1 indicate that the higher the maximum sequence size, the more coverage we have, but the bigger will be the MegaBlocks. For sequence sizes above 32 there is no coverage gain for the presented benchmarks. There is a significant

difference in the average size of MegaBlocks between a maximum sequence size of 16 and 32, but the average size in the latter case is still comfortably inside the usual size of blocks in approaches which implement small loops [6]. The coverage is an average for all benchmarks, when executing the binaries previously generated with the O3 flag. Regarding the three partitioning schemes previously presented, the results indicate that the MegaBlock with parameter 32 (i.e., considering MegaBlocks with up to 32 SuperBlocks) has the potential to represent a significant portion of program execution. For the considered benchmarks, on average, MegaBlocks with 10 or more iterations represent almost 90% of the total execution; MegaBlocks with 100 or more iterations represent more than 60% of the total execution. Since the MegaBlock has a single execution path, mapping this block to hardware may need less effort than mapping blocks with more complex control-flow.

Table 1. MegaBlock sizes with respect to the number of executed instructions. We present results when we only take into account MegaBlocks with 2 or more iterations, and MegaBlocks with 10 or more iterations. Note that we are not interested in mapping blocks which have only one iteration. The weighted average has into account the number of iterations each MegaBlock of a particular size has

Max Sequence Size	Minimum Size		Maximum Size		Weighted Average	
	2	10	2	10	2	10
1	4	4	106	106	7.5	7.4
2	4	4	106	106	7.5	7.4
4	4	4	106	106	7.5	7.4
8	4	4	783	106	9.0	8.8
16	4	4	783	309	10.4	10.3
32	4	4	783	309	26.0	25.6
64	4	4	783	309	26.0	25.6

Figure 4 presents a comparison between several partitioning methods using the 13 benchmarks. The execution of the programs was partitioned in blocks, and we measured, during the program execution, the number of consecutive iterations that occurred for each block. Besides the methods presented in Figure 2, we also implemented the partitioning method used by the Warp processor [6]. Note, however, that this last partitioning method detects complete loops with control-flow, while the others are biased, branchless paths. The curves in the figure should be seen relative to one another: they are particular for a set of benchmarks, and even the same program can present a different number of iterations with a different set of input data. For the considered benchmarks, the MegaBlock with a maximum pattern size of 32 is consistently above the other considered methods. This means that for the same number of iterations, the blocks found by this partitioning method represent a higher percentage of the executed code. It was expected that the Warp partitioning method could present a higher coverage, since it is covering not only the frequent path of the loop but also all the other paths of the loop. It seems that the method used by Warp [6] detects only inner loops. As the MegaBlock detects patterns of SuperBlocks, if an inner loop can fit in a small number of SuperBlocks and the size of the maximum sequence is sufficiently big, the MegaBlock partitioning method will automatically

consider small unrolled inner loops. The SuperBlock partitioning method follows the Warp partitioning method very closely. This is to be expected, since them both use backward branches to detect small loops.

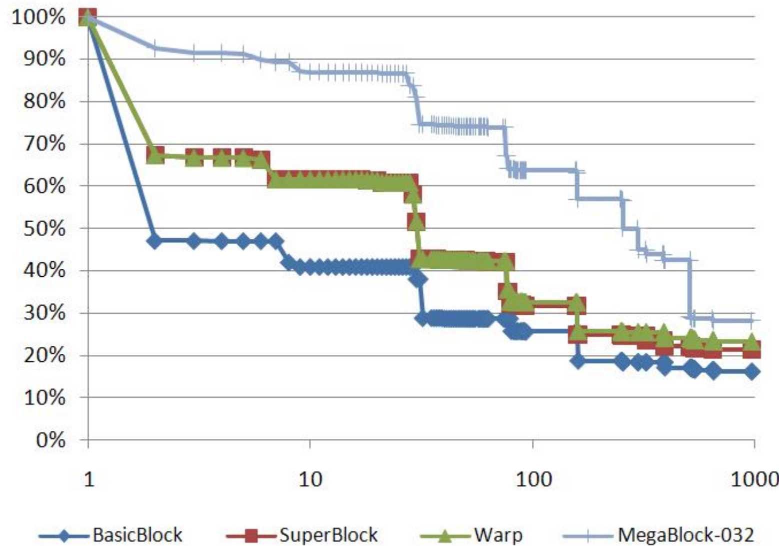

Figure 4. Portions of the program execution (*Y axis*) that are covered by blocks, identified by several partitioning techniques, which have *at least* a certain amount of iterations (*X axis*). We are using MegaBlocks which have, at most, 32 SuperBlocks

To analyze the potential speedups that can be obtained using our approach, we considered a hypothetical large 2D CGRA architecture coupled to the MicroBlaze. This CGRA consists of a number of rows, each one with functional units (FUs) and a single load/store unit. Each row is executed in one clock cycle. We consider that each instruction in the program execution can be mapped to a functional unit. We imposed a communication restriction, where FUs from a given row could only communicate with the FUs of the row immediately below. When an FU needed data from another FU from a distance higher than one row, the data items are communicated through other unoccupied FUs using "move" instructions. The only exceptions were data inputs, which can be read by any row. We also imposed restrictions for the memory operations: at any given row, there is only one load/store operation. This architecture is very similar in concept to the DIM architecture [19].

The mapping algorithm is based on the algorithm used by Clark *et al.* to map instructions to the CCA [20]. The MegaBlock is read as a stream of instructions, and each incoming instruction is placed on the first row which respects the data dependencies. After placement of the instruction, the algorithm checks if the instruction can receive the required data, and if not, inserts the necessary 'move' instructions. Additionally, it uses a conservative approach for memory instructions, mapping any load operation after the last store operation and respecting the occurrence order of store operations and possible side-exits. The speedup figures

account for communication overheads between reconfigurations, and assume we need one clock-cycle to communicate each live-in and live-out register.

Figure 5 shows the speedups for each benchmark across several levels of compiler optimization, when we use the MegaBlock partitioning method with a maximum sequence size of 32 and we move to hardware all blocks which have at least 10 iterations. As shown in Figure 5, we can achieve speedups from 2× to 4× on average, depending on the optimization level of the compiler. Higher optimization levels show higher speedups across most benchmarks, which come from a higher coverage rate for those optimization levels. This might happen because higher optimization levels can represent code in a more efficient format, which can benefit the pattern matching (e.g., less SuperBlocks for a given pattern). When there are simultaneously patterns of several sizes (e.g., the sequence AAAA has patterns of size one – A – and two – AA), since the pattern matching algorithm gives priority to the pattern with the smallest size, it is able to extract the smallest common kernel, even when the compiler uses optimization techniques which increase the size of the code (e.g., loop unrolling).

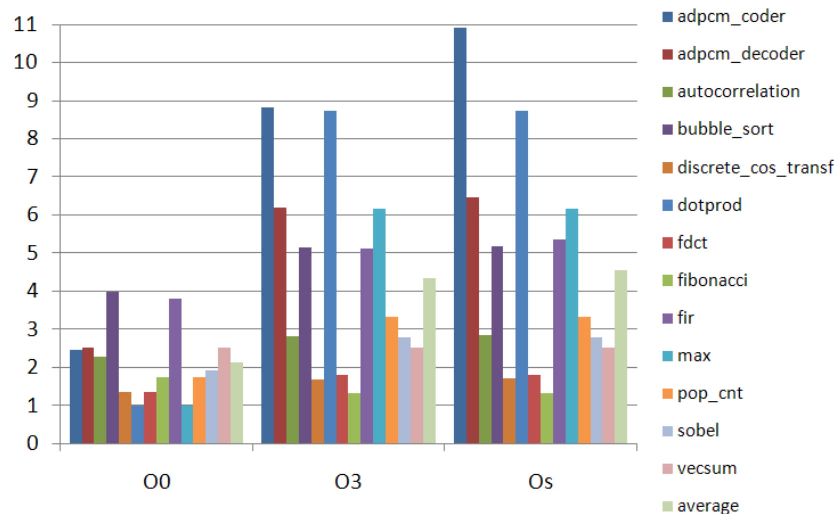

Figure 5. Speedups across different levels of compiler optimizations

5 Related Work

In the context of embedded systems, there have been several efforts addressing the dynamic mapping applications to RPUs.

Lysecky et al. [6] propose the Warp Processor, a system which includes a GPP, a fine-grained RPU and a dynamic mapping module. The dynamic mapping module automatically detects critical loops executing on the GPP and maps the corresponding binary code to the fine-grained RPU. In a posterior work [21], they use the same technique to improve the performance of a MicroBlaze *softcore* processor.

The Configurable Compute Accelerator (CCA) [22] is a special-purpose unit designed to be integrated in the pipeline of a GPP and executes a restrict set of Data-Flow Graphs (DFGs). The CCA cannot be directly accessed through programming, and instead, the unit itself has hardware support for binary translation, which automatically moves code from the instruction pipeline to the CCA.

Beck et al. [19] propose the Dynamic Instruction Merging (DIM) technique, a binary translation method to transparently map Basic Blocks from a general purpose MIPS processor to a coarse-grained reconfigurable array. They tightly couple the coarse-grained array to the processor, working as an additional functional unit in the execution stage of the pipeline. The objective of this architecture is to accelerate embedded systems that need to execute many different kinds of tasks.

Regarding these three approaches, Warp uses fine-grained reconfigurable hardware as the target RPU of dynamic mapping. Comparing to a coarse-grained RPU, it trades-off higher flexibility in the circuitry that can be implemented with higher mapping overhead. It is also an approach which needs a greater mapping effort, and that is not tightly coupled to the processor: both the CCA and the DIM are integrated in the pipeline of the processor, while the Warp RPU works as a co-processor. In the other hand, this enables the mapping of larger blocks in the Warp Processor. It implements complete loops, while the CCA and the DIM exploit ILP inside a small number of Basic Blocks.

6 Conclusions

This paper presented our approach to dynamically migrate computationally intensive sections of program execution from a general purpose processor to a coarse-grained reconfigurable array working as a co-processor. We proposed the MegaBlock for partitioning and presented experimental results showing a comparison between our approach and other common approaches such as the BasicBlock and the SuperBlock.

Ongoing work is addressing runtime optimizations and studying their impact in the final speedups. Future work will address hardware implementations of some of the modules needed to implement our dynamic mapping approach in order to quantify some of the resultant characteristics.

Acknowledgments

This research has been sponsored by the Portuguese Science Foundation (FCT) under research grants PTDC/EEA-ELC/70272/2006 and SFRH/BD/36735/2007.

References

[1] Z. Guo, W. Najjar, F. Vahid, and K. Vissers, "A quantitative analysis of the speedup factors of FPGAs over processors," in *FPGA '04: Proceedings of*

the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays, Monterey, California, USA, 2004, pp. 162-170.
[2] J. Henkel, "A low power hardware/software partitioning approach for core-based embedded systems," in *Annual ACM IEEE Design Automation Conference: Proceedings of the 36 th ACM/IEEE conference on Design automation*: Association for Computing Machinery, Inc, One Astor Plaza, 1515 Broadway, New York, NY, 10036-5701, USA, 1999.
[3] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, "The Transmeta Code Morphing™ Software: using speculation, recovery, and adaptive retranslation to address real-life challenges," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*: IEEE Computer Society Washington, DC, USA, 2003, pp. 15-24.
[4] G. Stitt, R. Lysecky, and F. Vahid, "Dynamic hardware/software partitioning: a first approach," in *Proceedings of the 40th conference on Design automation*: ACM New York, NY, USA, 2003, pp. 250-255.
[5] J. Bispo and J. M. P. Cardoso, "On Identifying Segments of Traces for Dynamic Compilation," in *20th International Conference on Field Programmable Logic and Applications (FPL'10)*, PhD Forum, Milano, Italy, 2010. (accepted)
[6] R. Lysecky, G. Stitt, and F. Vahid, "Warp Processors," *ACM Trans. Des. Autom. Electron. Syst.,* vol. 11, pp. 659-681, 2006.
[7] S. Hauck and A. DeHon, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*: Morgan Kaufmann/Elsevier, 2008.
[8] R. Hartenstein, "A decade of reconfigurable computing: a visionary retrospective," in *Proceedings of the conference on Design, automation and test in Europe*: IEEE Press Piscataway, NJ, USA, 2001, pp. 642-649.
[9] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix," in *Field-Programmable Logic and Applications*, 2003, pp. 61-70.
[10] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. C. Filho, "MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *Computers, IEEE Transactions on,* vol. 49, pp. 465-481, 2000.
[11] Z. A. Ye, A. Moshovos, S. Hauck', and P. Banerjee, "CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit," in *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, 2000, pp. 225-235.
[12] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt, "PACT XPP—A Self-Reconfigurable Data Processing Architecture," *The Journal of Supercomputing,* vol. 26, pp. 167-184, 2003.
[13] K. Bondalapati, P. Diniz, P. Duncan, J. Granacki, M. Hall, R. Jain, and H. Ziegler, "DEFACTO: A design environment for adaptive computing technology," 1999, pp. 570-578.
[14] Y. Yankova, G. Kuzmanov, K. Bertels, G. Gaydadjiev, Y. Lu, and S. Vassiliadis, "DWARV: DelftWorkbench Automated Reconfigurable VHDL

Generator," in *VHDL generator", the 17th International Conference on Field Programmable Logic and Applications (FPL'07*: Citeseer, 2007, pp. 697-701.
[15] I. Xilinx, "Microblaze processor reference guide," *reference manual,* 2006.
[16] W. M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, and G. E. Haab, "The superblock: an effective technique for VLIW and superscalar compilation," *The Journal of Supercomputing,* vol. 7, pp. 229-248, 1993.
[17] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: a transparent dynamic optimization system," in *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation* Vancouver, British Columbia, Canada: ACM, 2000.
[18] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "An efficient method of computing static single assignment form," in *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* Austin, Texas, United States: ACM, 1989.
[19] A. C. S. Beck, M. B. Rutzig, G. Gaydadjiev, and L. Carro, "Transparent reconfigurable acceleration for heterogeneous embedded applications," in *Proceedings of the conference on Design, automation and test in Europe* Munich, Germany: ACM, 2008.
[20] N. Clark, J. Blome, M. Chu, S. Mahlke, S. Biles, and K. Flautner, "An Architecture Framework for Transparent Instruction Set Customization in Embedded Processors," in *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, 2005, pp. 272-283.
[21] R. Lysecky and F. Vahid, "Design and implementation of a MicroBlaze-based warp processor," *ACM Transactions on Embedded Computing Systems,* vol. 8, 2009.
[22] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner, "Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization," in *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture* Portland, Oregon: IEEE Computer Society, 2004.