Performance Estimation Framework for Automated Exploration of CPU-Accelerator Architectures

Tobias Kenter
Paderborn Center for Parallel Computing
University of Paderborn, Germany
kenter@upb.de

Christian Plessl
Paderborn Center for Parallel Computing
University of Paderborn, Germany
christian.plessl@upb.de

Marco Platzner
Computer Engineering Group
University of Paderborn, Germany
platzner@upb.de

Michael Kauschke
Intel Microprocessor Technology Lab
Braunschweig, Germany
michael.kauschke@intel.com

ABSTRACT

In this paper we present a fast and fully automated approach for studying the design space when interfacing reconfigurable accelerators with a CPU. Our challenge is, that a reasonable evaluation of architecture parameters requires a hardware/software partitioning that makes best use of each given architecture configuration. Therefore we developed a framework based on the LLVM infrastructure that performs this partitioning with high-level estimation of the runtime on the target architecture utilizing profiling information and code analysis. By making use of program characteristics also during the partitioning process, we improve previous results for various benchmarks and especially for growing interface latencies between CPU and accelerator.

Categories and Subject Descriptors

B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids; I.6.m [**Simulation and Modeling**]: Miscellaneous

General Terms

Performance, Design, Algorithms

1. INTRODUCTION

Reconfigurable hardware accelerators promise to improve performance and energy efficiency over conventional CPUs for a wide range of applications. The integration of CPU and accelerator is an essential design decision since the characteristics of the chosen interface have a major impact on the granularity of functions that can be offloaded to the accelerator, on feasible execution models, and on the achievable performance or performance/power benefits. Related work has proposed numerous approaches for this interface [1], such as accelerators integrated as functional units into a CPU's data-path or attached to a CPU via the co-processor interface, the multi-processor interconnect, the memory subsystem, or an IO bus. In this work we focus on a subclass of CPU-accelerator architectures, where the accelerator is embedded into the compute system through two interfaces as illustrated in Figure 1. First a direct low latency interface to an adjacent general purpose CPU allows fine grained interaction mainly on the control level like activating a particular configuration of the accelerator, triggering its execution and synchronizing with its results. The second interface gives the accelerator access to the memory hierarchy independently from the CPU.

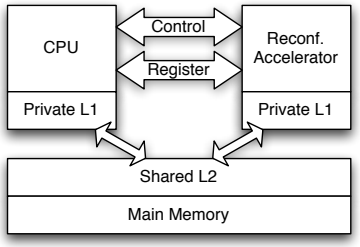

Figure 1: Architecture with shared L2 and two private L1 caches

Performance estimation and design space exploration for this and other classes of CPU-accelerator architectures are challenging problems. Simulation is the most common approach to evaluate the architectural integration of reconfigurable accelerators before prototyping. The time-consuming design process of simulation or co-simulation often limits it to assume a specific interface and a hardware/software partitioning that is hand tailored to the characteristics of this interface. The challenge for an automated design space exploration is that the specifications of the interface affect what parts of the application can be mapped to the accelerator during hardware/software partitioning.

The contribution of this paper is a new approach for fast and fully automated performance estimation of CPU-accelerator architectures. By combining high-level analytical performance modeling, code analysis and profiling and automated hardware/software partitioning we can estimate the achievable speedup for arbitrary applications executing on a

FPGA'11, February 27–March 1, 2011, Monterey, California, USA.

Published in: *Proc. Int. Symp. on Field-Programmable Gate Arrays (FPGA)*
© ACM 2011, DOI: 10.1145/1950413.1950448

Table 1: Data computed by the framework

Execution count of instruction I_k	$n(I_k)$
Count of control flow from B_l to B_m	$n(B_l, B_m)$
Mapping of basic block or instruction	$p(B_l)$, $p(I_k)$
Cache level that executes a load/store	$v(I_k^j)$
Register value used or produced by B_l	$R_u(B_l)$, $R_w(B_l)$

wide range of CPU-accelerator architectures. The intended use of our method is to quickly identify the most promising areas of the large CPU-accelerator design space for subsequent in-depth analysis and design studies. Consequently, we emphasize modeling flexibility and speed of exploration rather than a high accuracy of the estimation method. The main benefit of our method is that it needs only the application source code or LLVM binary and does not require the user to extract any application-specific performance parameters by hand.

We have introduced the basic performance estimation approach underlying this work in [3]. This preliminary work however showed limitations in the greedy block based hardware/software partitioning strategy that was susceptible to get stuck in local minimal. Here, we present an improved multi level partitioning algorithm that uses loop and function information gained with code analysis which yields significantly better results for some benchmarks and increases the overall robustness in the presence of longer communication latencies. Further, we provide a more comprehensive presentation of our method and evaluate it with a broader range of benchmarks. Among known high level estimation methods, the approach of Spacey et al. [5] is most closely related to our work. There are, however, three differences. First, in contrast to the estimation method of Spacey et al. which models the memory subsystem with a single bandwidth parameter, our framework includes cache models and thus more realistically mimics relevant architectures. Second, their partitioning approach is limited to the block level while we show our new multi level partitioning technique to be clearly inferior. Finally, whereas the system of Spacey et al. is x86 assembly based and can therefore be applied to x86 binary code, our framework leverages the LLVM infrastructure [4] which will allow us to extend the framework towards automated code generation for various targets.

2. ESTIMATION METHOD

In this section, we present the basic terms and definitions of our model and how they are combined into the total runtime estimation. We furthermore describe how the analysis framework gathers data of the execution characteristics of a program as shown in Table 1. The architecture parameters of our model are also introduced here. They are summarized along with their default values in Table 2.

Our estimation framework is based on the LLVM compiler infrastructure [4]. The investigated software is compiled into LLVM assembly language, which is the intermediate code representation on which LLVM's analysis and optimization passes work. We model a program as a set of instructions $I = \{I_1, I_2, \ldots, I_{\texttt{nins}}\}$, and classify the instructions into memory dependent operations (e. g., load/store) and independent operations: $I_k \in \{\texttt{ld/st}, \texttt{op}\}$. The program structure groups the instructions into a set of basic blocks $B = \{B_1, B_2, \ldots, B_{\texttt{nblks}}\}$. The basic blocks form a control flow graph where an edge $B_l \rightarrow B_m$ denotes that block B_m might be executed directly after block B_l. The instructions of B_l use a set of register values $R_u(B_l)$ and produce a set of register values $R_w(B_l)$.

The architecture comprises two processing units, the CPU and the accelerator (ACC), and a memory hierarchy consisting of L1, L2 and optional L3 caches, and main memory (MEM). The caches can be private for each core like the L1 cache in Figure 1 or shared between both units like the L2 cache in the same figure. We model the execution efficiencies $\epsilon(\texttt{CPU})$ and $\epsilon(\texttt{ACC})$ of the cores through the average number of clock cycles spent per instruction. The efficiency reflects on the one hand raw execution times of instructions and on the other hand parallel execution units and pipelining effects, which increase the throughput. The efficiencies can vary for different instruction classes, but lacking more accurate data for both cores, in this work we use identical efficiency for all instructions except for two instruction classes. First, we assume that LLVM typecast instructions are implemented on the accelerator through wiring and thus are executed in zero execution time. Second, load/store instructions are also considered in a different way, because their execution time depends on the level in the memory hierarchy that they access. We describe the corresponding access latencies with $\lambda_m(\texttt{L1})$, $\lambda_m(\texttt{L2})$ and $\lambda_m(\texttt{MEM})$, expressed in clock cycles. When a core requires data resident in the private cache of the other core we include a latency penalty for writing back the data to the shared cache.

For communication between the CPU and the accelerator, we define λ_c as latency for transferring control between the cores. This control latency also covers the efficiency losses that may occur, when the pipelining of instructions is reduced by control changes. Furthermore, we denote λ_r as latency for transferring a register value. Refining the register value transfer model, we foresee a push method with a low latency of $\lambda_{r,\texttt{push}}$ for actively sending a register value from one location to the other and a somewhat slower pull method with latency $\lambda_{r,\texttt{pull}}$ for requesting a register value from the other location and receiving it. Since the analysis of register dependencies is based on the code in LLVM intermediate representation, no register allocation has taken place, so all values are treated as available in an infinite register file after their first occurrence.

The partitioning process maps each basic block to either the CPU or the accelerator. We denote the mapping of block B_l as $p(B_l)$ with the two possible values $p(B_l) = \texttt{CPU}$ or $p(B_l) = \texttt{ACC}$. Obviously, the partitioning of basic blocks also implies a partitioning $p(I_k)$ of instructions I_k into $p(I_k) = \texttt{CPU}$ or $p(I_k) = \texttt{ACC}$, since $\forall k, l : I_k \in B_l \rightarrow p(I_k) = p(B_l)$. At this time we do not assume a concurrent execution on both CPU and accelerator, so the execution order of the program remains unchanged regardless of the mapping. Our model limits the total number of instructions mapped to the accelerator and accounts for a unit area for each such instruction. The use of an averaged area value is most reasonable when the internal architecture of the accelerator is a coarse grained array of homogenous functional units.

Using the LLVM infrastructure to profile program executions, we determine the execution count for an instruction I_k as $n(I_k)$, for a basic block B_l as $n(B_l)$, and the number of control flows over an edge $B_l \rightarrow B_m$ of the control flow graph as $n(B_l, B_m)$. Furthermore, we denote the jth execution of instruction I_k as I_k^j and use this separation to determine the

level accessed in the memory hierarchy for each load/store instruction as $v(I_k^j)$, with the possible values `L1`, `L2` and `MEM`. To determine these values $v(I_k^j)$, we add a memory profiling pass to LLVM and perform a simulation of an inclusive, direct mapped cache hierarchy. These characteristics of the cache enables us, together with a data dependency analysis, to compute the memory access time for each partitioning step without a repeated cache simulation, which would otherwise slowdown the partitioning process significantly. At this point we also profit from the missing register allocation in that no register spills occur, which would change the memory access patterns for different mappings.

We estimate the total program runtime as the sum of four components: the **execution time** t_e of instructions with only register operands, the **memory access time** t_m for load and store instructions, the **control transfer time** t_c for switching control between successive basic blocks mapped to different cores, and the time **register value transfer time** t_r:

$$t = t_e + t_m + t_c + t_r$$

$$t_e = \sum_{k:I_k=\texttt{op}} n(I_k) \cdot \epsilon(p(I_k))$$

$$t_m = \sum_{k:I_k=\texttt{ld/st}} \sum_{j=1}^{n(I_k)} \lambda_m(v(I_k^j))$$

$$t_c = \sum_{(l,m)} n(B_l, B_m) \cdot \lambda_c$$

$$\forall (l,m) : (B_l \rightarrow B_m) \wedge p(B_l) \neq p(B_m)$$

$$t_r = \sum_{R} \min\left(n(B_l) \cdot \lambda_{r,\texttt{push}}, n(B_m) \cdot \lambda_{r,\texttt{pull}}\right)$$

$$\forall R : R \in R_w(B_l) \wedge R \in R_u(B_m) \wedge p(B_l) \neq p(B_m)$$

3. PARTITIONING APPROACH

We utilize a greedy partitioning algorithm that starts with all blocks at the CPU and iteratively moves partitioning objects *po* to the accelerator, as long as the cumulative area of all moved blocks fits the size of the accelerator. An overview of the partitioner is given in Algorithm 1. At each step, the partitioning object with the highest attractiveness, which is the ratio of estimated speedup to area requirements, is chosen in the function *getBestPartitionObject*. We compare two versions of the partitioning approach, which differ by the definition of their partitioning objects. For the basic block level partitioning, we utilize all single basic blocks as partitioning objects. For the improved multi level partitioning, we use not only all single basic blocks as partitioning objects but additionally all loops (inner as well as nested loops) and all functions. Furthermore, whenever the partitioner moves part of a loop or function to the accelerator, the remaining basic blocks of the loop or function form another new partitioning object. This definition of partitioning objects is a heuristic covering the most promising objects of the total space of potential partitioning objects which grows exponentially with the number of basic blocks.

By this extension of the simple block level partitioning method which we have used in our earlier work, we were able to achieve significantly better results in cases when moving a single basic block lead to a local maximum in the cost function preventing detection of a minimum only reachable by moving a collection of basic blocks like those representing a loop. Figure 2 illustrates an example where this may occur. Let's assume that every single execution of each individual basic block B_1 to B_3 can save one clock cycle when it is executed on the accelerator instead of the CPU. Let furthermore each control flow between two blocks that are assigned to different processing units lead to a communication overhead of also one clock cycle. Now neither B_2 nor B_3 will provide a speedup when individually moved to the accelerator, which is what will be tried by the block level partitioning method. Since the two basic blocks B_1 and B_2 form a loop the new multi level partitioning will also attempt to move both loop blocks at once to the accelerator, which yields a speedup in the example.

Input: Resources of Accelerator, All BasicBlocks mapped to CPU
Output: Partitioned BasicBlocks
Next ← getBestPartitionObject(Resources);
while *Next* ≠ *null* **do**
  moveToAccelerator(Next);
  Resources ← Resources - computeArea(Next);
  Next ← getBestPartitionObject(Resources);
end

Algorithm 1: Algorithm of our Partitioning Approach

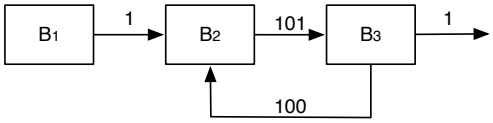

Figure 2: Example of a control flow graph where the basic blocks forming a loop might not be moved to the accelerator with block level partitioning (edges labeled with execution frequency)

4. RESULTS

We evaluate our performance estimation framework with 13 benchmarks, most taken from the MiBench suite [2] and selected to represent a wide range of application classes. The architecture parameters for the tests are summarized in Table 2. They allow a speedup of up to factor 2 in execution times, but offers at best identical memory access times compared to a CPU only solution.

In Figure 3, we give an overview of the expected speedups for each individual benchmark with both partitioning approaches. It comes as no surprise, that speedup potential varies a lot depending on the application characteristics. According to the multi level partitioning results, it ranges

Table 2: Default model parameters

Execution efficiencies		Cache sizes	
ϵ(`CPU`)	1.0 cycles	L1	32KB
ϵ(`ACC`)	0.5 cycles	L2	4MB
Communication latencies		Cache latencies	
λ_c	2 cycles	λ(`L1`)	3 cycles
$\lambda_{r,push}$	1 cycles	λ(`L2`)	15 cycles
$\lambda_{r,pull}$	3 cycles	λ(`MEM`)	200 cycles
Accelerator size: 2048 units			

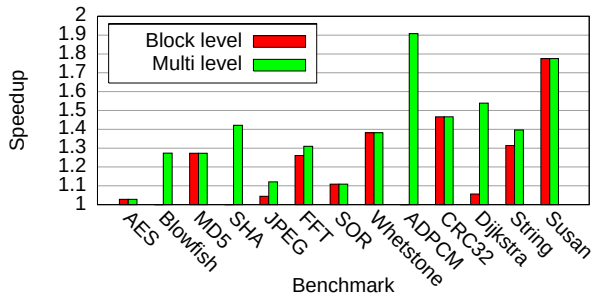

Figure 3: Speedups for 13 benchmarks at accelerator size 2048 for both partitioning approaches

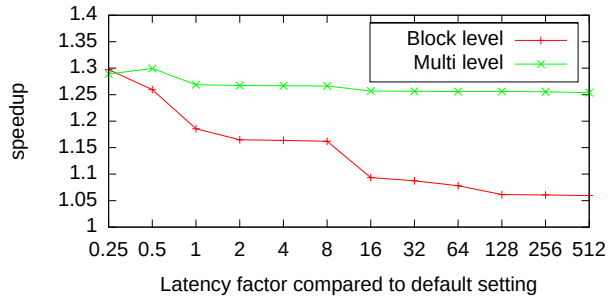

Figure 4: Speedups for different interface latencies, averaged over 13 benchmarks, with block level and multi level partitioning

from 1.03 to 1.91, with the median at 1.38. When studying the effect of multi level partitioning compared to our previously utilized block level partitioning, the new approach performs consistently better or identical to the older one. The differences of individual results reflect to which degree the benchmarks contain loops that are responsible for a large percentage of the total execution time and have large communication requirements if split between CPU and accelerator. For example we observe that for ADPCM with default parameters, a loop with seven basic blocks is moved to the accelerator as first partitioning object.

Most remarkable are the effects of multi level partitioning when regarding the robustness for growing interface latencies in Figure 4. With a hypothetical minimal latency, the block level partitioning performs actually better, because here the multi level partitioning gets stuck in a local optimum for the FFT benchmark, whereas all other benchmarks show almost identical results between both approaches. But then the block level partitioning rapidly looses lots of performance, whereas the multi level partitioning can retain around 85% of its maximum speedups. This observation underlines that longer interface latencies increase the need to place basic blocks with relevant communication requirements together at one component, for which the multi level partitioning method is better suited.

Similarly to changing the interface latencies, we can vary the other specified architecture parameters individually or in groups and also investigate different configurations for the architectural integration of the accelerator into the memory hierarchy, enabling design space exploration. Due to space limitations we can not present details here, for early results with the block level partitioning algorithm refer to [3]. A central result of the former work has been confirmed with the new partitioning algorithm and the 13 benchmarks presented here. It indicates, that a sharing a L1 cache between CPU and accelerator is the best way to integrate the two cores, if there is no penalty for the increased complexity of a shared cache over a private one. However, once we assume an increased latency of only 1 cycle in order to reflect this increased complexity, the picture changes. Under this premise, a shared L2 cache with private L1 caches for both CPU and accelerator turns out as the best memory hierarchy after a penalty for the shared cache is applied.

5. CONCLUSION AND FUTURE WORK

This paper presents our estimation and hardware/software partitioning framework, which yields significantly better partitioning results than our previous work. The capability of our framework to provide fully automated estimation and partitioning results can be used for a systematic design space exploration that was previously hard to undertake for new architectures.

For assessing the absolute quality of the partitioning results found with our multi level partitioning, we plan to compare them to optimal results that we hope to obtain by formulating the design space exploration as an integer linear programming problem which can be solved exactly with appropriate solvers. We also intend to cover the register allocation and to refine the latency based communication and memory latency models, and to evaluate the refined models by applying them to existing CPU-accelerator architectures.

Acknowledgment

This work is supported by Intel Corporation through a grant for the project “A multimode reconfigurable processing unit (MM-RPU)”.

6. REFERENCES

[1] K. Compton and S. Hauck. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys*, 34(2):171–210, June 2002.

[2] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *WWC ’01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[3] T. Kenter, M. Platzner, C. Plessl, and M. Kauschke. Performance estimation for the exploration of CPU-accelerator architectures. In O. Hammami and S. Larrabee, editors, *Proc. Workshop on Architectural Research Prototyping (WARP)*, June 2010.

[4] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. 2004 Int. Symp. on Code Generation and Optimization (CGO)*, pages 75–86. IEEE Computer Society, Mar 2004.

[5] S. A. Spacey, W. Luk, P. H. J. Kelly, and D. Kuhn. Rapid design space visualization through hardware/software partitioning. In *Proc. Southern Programmable Logic Conference (SPL)*, pages 159–164. IEEE, Apr. 2009.