Performance Estimation for the Exploration of CPU-Accelerator Architectures

Tobias Kenter, Marco Platzner and Christian Plessl
University of Paderborn
Paderborn, Germany
Email: kenter@uni-paderborn.de

Michael Kauschke
Intel Microprocessor Technology Lab
Braunschweig, Germany

***Abstract*—In this paper we present an approach for studying the design space when interfacing reconfigurable accelerators with a CPU. For this purpose we introduce a framework based on the LLVM infrastructure that performs hardware/software partitioning with runtime estimation utilizing profiling information and code analysis. We apply it to reconfigurable accelerators that are controlled by a CPU via a direct low-latency interface but also have direct access to the memory hierarchy. Our results show that a shared L2 cache for CPU and accelerator seems to be the most promising design point for a range of applications.**

I. Introduction and Related Work

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. 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 our work the accelerator is embedded into the compute system through two interfaces. First a direct low latency interface to an adjacent general purpose CPU allows fine grained interaction mainly on control level like steering, synchronization and reconfiguration of the accelerator. This may also include exchange of small data entities like predicates, flags and small scalars. The second interface gives the accelerator access to the memory hierarchy independent from the CPU. It may be tailored to the needs of the accelerator without compromising the CPU architecture, thus simplifying the overall design. Figure 1 illustrates an example. The distinctive feature of this coupling is the support of a wide range of granularities of accelerated code parts, reaching from custom instructions via kernel loops to functions or threads. Although the proposed interface does not require a complete CPU redesign, it still allows for the acceleration of fine grained tasks and, thus, increases single thread performance, an area where current CPU designs face diminishing returns.

Performance estimation and design space exploration for CPU-accelerator architectures are challenging problems. Simulation is the most common approach to evaluate the architectural integration of reconfigurable accelerators before prototyping. For example, Garcia et al. [2] rely on co-simulation to evaluate an architecture where CPU and accelerator work on the same memory hierarchy. While such a pure co-simulation approach provides some insight, its time-consuming design process often limits it to assume a specific interface and hardware/software partitioning. The challenge for an automated design space exploration is that the characteristics of the interface affect what parts of the application can be mapped to the accelerator during hardware/software partitioning. We consider this interdependency between interface and partitioning the reason why the systematic exploration of the design space for the architectural integration has so far not received significant attention in academic research.

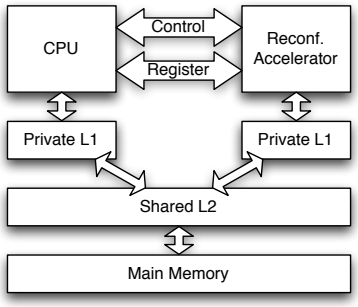

Fig. 1. Architecture with shared L2 and two private L1 caches

The main contribution of this paper is a fast and fully automated performance estimation method for assessing the speedup potential of various CPU-accelerator architecture design points. Among known high level estimation methods, e.g. [3], the approach of Spacey et al. [4] is most closely related to our work. In contrast to the x86 assembly based estimation method of Spacey et al., our framework includes cache models and leverages the LLVM infrastructure [5], which will allow us to extend the framework towards automated code generation for various targets.

II. Method and Framework

In this section we present our estimation method together with some aspects of the data generation before giving an overview of its limits that are to be addressed in future work.

A. Estimation Method

Our estimation framework is based on the LLVM compiler infrastructure. 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 load/stores and operations: $I_k \in \{\texttt{ld/st}, \texttt{op}\}$. The instructions are grouped 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 cores, the CPU and the accelerator (ACC), and a memory hierarchy consisting of private L1 caches, shared L2 cache, and memory (MEM). 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. Load/store instructions access a certain level in the memory hierarchy. We describe the corresponding access latencies as with $\lambda_m(\texttt{L1}), \lambda_m(\texttt{L2})$ and $\lambda_m(\texttt{MEM})$, expressed in clock cycles. For communication between the CPU and the accelerator, we define λ_c as latency for transferring control between the cores and 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 $\lambda_{r,\texttt{push}}$ and a somewhat slower pull method with latency $\lambda_{r,\texttt{pull}}$.

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 $\forall k, l : I_k \in B_l \rightarrow p(I_k) = p(B_l)$.

Using the LLVM compiler 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 $\texttt{L1}, \texttt{L2}$ and MEM. To determine these values $v(I_k^j)$, we added a memory profiling pass to LLVM and perform a simulation of an inclusive, direct mapped cache hierarchy that enables us together with a data dependency analysis to compute the memory access time for each partitioning step without a repeated cache simulation.

We estimate the total program runtime as 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 time t_c for transferring control between successive basic blocks mapped to different cores, and the time t_r for exchanging register values between CPU and accelerator:

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

Execution time:

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

Memory access time:

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

Control transfer time:

$$t_c = \sum_{(l,m)} n(B_l, B_m) \cdot \lambda_c$$
$$\forall (l,m) : (B_l \rightarrow B_m) \wedge p(B_l) \neq p(B_m)$$

Register value transfer time:

$$t_r = \sum_R \min\left(n(B_l) \cdot \lambda_{r,\texttt{push}}, n(B_m) \cdot \lambda_{r,\texttt{pull}}\right)$$
$$\forall R : R \in R_w(B_l) \wedge R \in R_u(B_m) \wedge p(B_l) \neq p(B_m)$$

B. Limitations of the Method

Our method contains some aspects that need to be refined in future work. The presented latency based approach neglects benefits of modern CPU's parallel memory accesses that result from superscalar load/store units, pipelining and out-of-order execution and speculative prefetching. Yet we assume that an accelerator working on rather regular tasks can achieve at least a similar effectivity for its memory interface. Therefore the method is inaccurate in that it overestimates the memory access time t_m, but it does not favor the accelerator.

Another aspect that is neglected in our framework is the register allocation, which has not taken place in the LLVM intermediate code. Hence register spills caused by capacity limits are missing in our estimation of the memory access time t_m and would also change the communication patterns assumed by our register value transfer time t_r.

We utilize a greedy partitioning algorithm that iteratively moves basic blocks with the highest ratio of estimated speedup to area requirements to the accelerator, as long as the cumulative area of all moved blocks fits the size of the accelerator. We observe, that while our partitioning produces reasonable results, the greedy approach fails, when several basic blocks, e.g. of a complex loop, would have to be moved together to the accelerator before any speedup occurs.

III. Results

The results presented in this section have been obtained based on the estimation method of Section II and two refinements. First, 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. Second, we added the option to differentiate between instruction classes and use this feature to assume cast instructions to be implemented on the accelerator through wiring and thus executing in zero time.

All presented experiments use the default parameters listed in Table I, unless stated otherwise. The default memory hierarchy is a configuration with shared L2 and private L1 caches

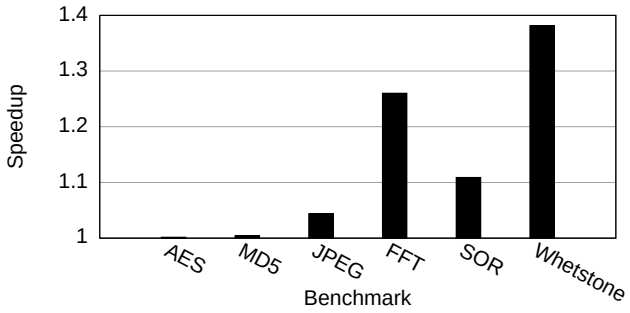

Fig. 2. Speedups for six benchmarks on the architecture specified in Table I

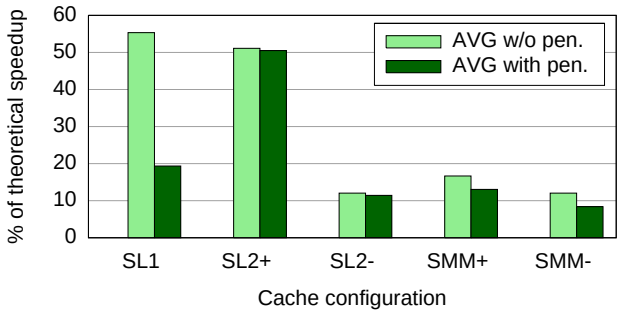

Fig. 3. Relative speedups for different memory integrations with and without a 1 cycle latency penalty for shared caches, averaged over six benchmarks

as depicted in Figure 1. The accelerator size is expressed by the number of LLVM instructions that can be mapped to the reconfigurable hardware core.

TABLE I
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: 128			

A. Speedups

We evaluate our performance estimation framework with six benchmarks that represent compute intense kernels from various application domains: The block cipher AES and the hash function MD5 from cryptography, JPEG as multimedia application, FFT and Successive Over-relaxation (SOR) kernels as representatives of scientific computing applications and, finally, the Whetstone benchmark. Figure 2 shows the resulting overall speedups for these benchmarks, which range from 0.16% to 38.16%. The main reason for the poor speedups of the cryptographic benchmarks is that our estimation method does not capture the bit level parallelism that is typically exploited to accelerate cryptographic functions on reconfigurable hardware. Furthermore, the cryptographic benchmarks suffer from high memory access times t_m compared to the execution times t_e and, in the case of *AES*, from an insufficient accelerator size.

In the following, we use as an evaluation metric the achieved percentage of the theoretical speedup, also denoted as relative speedup. The theoretical speedup merely serves as a baseline for comparing architecture alternatives and is computed by mapping all instructions to the accelerator, neglecting any resource limitations, communication latencies, increased memory latencies, and the fact that a practical accelerator will not be able to execute all instruction classes.

B. Memory integration

We investigate the impact of different models for integrating the accelerator into the memory hierarchy. The analyzed design points are: shared L1 data cache (SL1), shared L2 cache where both CPU and accelerator have private L1 caches (SL2+) (shown in Figure 1), shared L2 cache where only the CPU has a private L1 cache and the accelerator has no local memory at all (SL2-), shared main memory with private caches for both components (SMM+), and shared main memory with a private cache only for the CPU (SMM-). In order to account for the increased complexity of a shared cache over a private one, we also investigate how much of the speedup remains after applying a penalty by increasing the latency of the shared cache by 1 cycle. Figure 3 shows the results averaged over all six benchmarks.

We note that without penalty, a shared L1 cache (SL1) delivers the best performance. However, this architectural design point turns out to be highly sensitive to latency penalties. Overall, the shared L2 cache with private L1 caches for both CPU and accelerator (SL2+) is a well-performing and also robust design point since it retains most of its speedup potential when applying the latency penalty to the shared L2 cache. Design points with shared main memory as well as those where the accelerator does not have a private cache show similar performance which, however, is much lower than the performance for SL2+. While this experiment underlines the necessity for the accelerator to have access to low latency memory, it is not proven that this memory actually needs to be a cache. We leave it for further work to investigate alternative models for local accelerator memory.

C. Accelerator size

Figure 4 shows that the useful size of the accelerator strongly depends on the application. We can make two observations. First, for the individual benchmarks there are different regions in the accelerator size for which the speedups increase more slowly or more rapidly. Whenever the accelerator becomes large enough to utilize beneficial mappings of basic blocks, we can observe a steep increase in the relative performance. In Figure 4, this effect is visible for AES for accelerator sizes between 512 and 2048 and for FFT for accelerator sizes between 32 and 128, respectively. Second, some benchmarks do not show a saturation effect when scaling the accelerator size in the considered range. Since manufacturing limits and costs certainly constrain the

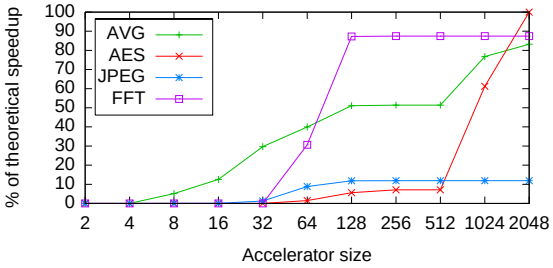

Fig. 4. Relative speedups for different accelerator sizes

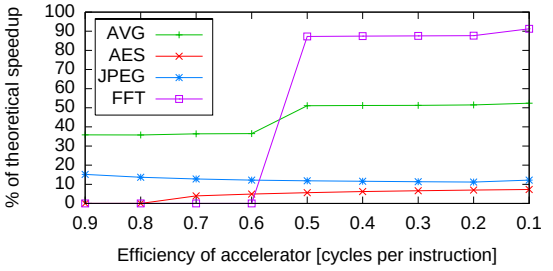

Fig. 5. Relative speedups for different accelerator execution efficiencies

maximum size of an accelerator, one should investigate temporal partitioning in tandem with rapid reconfiguration to further improve performance.

D. Execution efficiency

Figure 5 presents the relative performance over the execution efficiency of the accelerator. We have to note that in this illustration the baseline theoretical speedup grows with the accelerator efficiency. We observe that even at moderate accelerator execution efficiencies it is possible to identify basic blocks that can be mapped to reconfigurable hardware with a low enough overhead to achieve a speedup. For the JPEG benchmark it turns out that the relative performance decreases when the execution efficiency improves from 0.9 to 0.2. This results from the significant number of typecast instructions, which get mapped to the accelerator at zero execution time. When improving the accelerator's execution efficiency, the typecasts can not contribute to higher speedups while the theoretical speedup, depending to a larger part on register instructions, further increases with the execution efficiency.

E. Interface latency

Figure 6 depicts the relative performance over the latency of the interface between CPU and accelerator. We apply one scaling factor to all three parameters of the interface latency, $\lambda_c, \lambda_{r,pull}$ and $\lambda_{r,push}$, and investigate both lower and higher latencies than the default settings of Table I. Even though it might be architecturally infeasible to reach the lowest studied latencies, the experiment provides insights into the nature of the design space. Interestingly, for the JPEG benchmark we observe very similar partitioning results, which keep a significant portion of their speedup potential as the latency factor increases from 0.5 to 64 (corresponding to $\lambda_c = 1$ to $\lambda_c = 128$). At a latency factor of 0.25 a different partitioning with a much higher speedup becomes possible. The AES and FFT benchmark show a different behavior. With the given design parameters these benchmarks require very low latencies to achieve speedups at all.

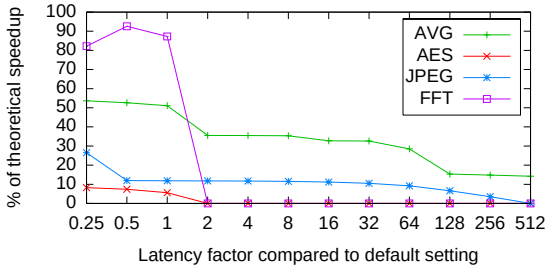

Fig. 6. Relative speedups for different interface latencies

IV. Conclusion

This paper presents our estimation and hardware/software partitioning framework that we utilize to investigate architectural integration of accelerators to CPUs. The results show that considering a latency penalty for designing a shared cache, the best design point is to share the L2 cache between CPU and accelerator. We also point out that some sort of local memory is important for an accelerator to achieve speedups. Furthermore our tests show, that some speedups can already been achieved with moderate improvements of the accelerator for the execution efficiency or with a rather slow communication interface. Yet more optimistic design parameters can lead to significantly better speedups by enabling new partitioning results that don't pay of otherwise.

Our framework produces fully automated estimation and partitioning results. As it is based on the LLVM infrastructure, it is possible to utilize and extend LLVM's optimization and code generation features in order to build complete tool flow to compile software for the proposed architecture.

Acknowledgment

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

References

[1] K. Compton and S. Hauck, "Reconfigurable computing: A survey of systems and software," *ACM Computing Surveys*, vol. 34, no. 2, pp. 171–210, Jun. 2002.

[2] P. Garcia and K. Compton, "A reconfigurable hardware interface for a modern computing system," in *Proc. 15th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*. IEEE Computer Society, Apr. 2007, pp. 73–84.

[3] B. Holland, K. Nagarajan, and A. D. George, "RAT: RC amenability test for rapid performance prediction," *ACM Trans. on Reconfigurable Technology and Systems*, vol. 1, no. 4, pp. 1–31, 2009.

[4] 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)*. IEEE, Apr. 2009, pp. 159–164.

[5] 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)*. IEEE Computer Society, Mar 2004, pp. 75–86.