Optimized High-Level Synthesis of SMT Multi-Threaded Hardware Accelerators

Jens Huthmann*, Andreas Koch*
*Embedded Systems and Applications Group (ESA), Technische Universität Darmstadt, Germany
Email: {jh,ahk}@esa.cs.tu-darmstadt.de

***Abstract*—**
Recent high-level synthesis tools offer the capability to generate *multi-threaded* micro-architectures to hide memory access latencies. In many HLS flows, this is often achieved by just creating multiple processing element-instances (one for each thread). However, more advanced compilers can synthesize hardware in a *spatial form* of the barrel processor- or simultaneous multi-threading (SMT) approaches, where only state storage is replicated per thread, while the actual hardware operators in a single datapath are re-used between threads. The spatial nature of the micro-architecture applies not only to the hardware operators, but also to the *thread scheduling facility*, which itself is spatially distributed across the entire datapath in separate hardware stages. Since each of these thread scheduling stages, which also allow a re-ordering of threads, adds hardware overhead, it is worthwhile to examine how their number can be reduced while maintaining the performance of the entire datapath. We report on a number of thinning options and examine their impact on system performance. For kernels from the MachSuite HLS benchmark collection, we have achieved area savings of up to 50% LUTs and 50% registers, while maintaining full performance for the compiled hardware accelerators.

I. Introduction

High-level synthesis tools translating different subsets of C into synthesizable HDL code are under active development from many commercial vendors and academic groups alike. Commercial tools include Xilinx Vivado HLS [1], Y Explorations eXCite [2], and Synopsis Synphony C Compiler [3]. These tools, however, do not perform co-compilation into hybrid hardware/software-executables, which is still the domain of a small number of academic projects such as LegUp [4], ROCCC [5], Comrade [6], and DWARV [7]. The topic of exploiting multi-threaded execution to hide memory access latencies in the generated hardware is even more rarely addressed.

NYMBLE-MT [8] is a specialized back-end for the NYMBLE hardware compilation system [9]. In addition to hardware-software co-compilation from C to shared-memory heterogeneous reconfigurable computers, the MT back-end is able to generate *multi-threaded* accelerators following the barrel processor or SMT approaches. In these micro-architectures, only state storage is replicated per thread, while the actual compute operators are shared between threads. This is achieved by selectively applying coarse-grained dynamic scheduling and per-thread context data storage. The aim here is to exploit these mechanisms not just to hide memory access latencies, but to improve datapath usage in the presence of Variable-Latency Operations (VLO) in general. Note that in this model, nested loops having variable execution times (e.g., due to variable loop bounds or internal control flow) are also considered VLOs, making multi-threading even more applicable.

978-1-4673-9091-0/15/$31.00

Section III gives a brief overview over some of the issues that need to be addressed in NYMBLE-MT. Note that the key paradigm of reconfigurable computing, namely spatially distributing the computation, is also followed for the scheduling facility itself. It is realized as multiple independent Hardware Thread Scheduling (HTS) stages, which are transparently inserted into the datapath.

This work presents two main contributions over the original NYMBLE-MT research: First, it examines the potential for improving the area efficiency of the multi-threaded accelerators by removing HTS stages from the datapath, introducing the new concepts of *mandatory* and *optional* HTS stages. Second, it presents two heuristics for the selective HTS removal process (one using a rule-guided brute-force approach, the second relying on dynamic profiling of datapath behavior).

II. Related Work

In [10], the CHAT compiler is introduced as a variant of ROCCC capable of generating multi-threaded accelerators that allow for a very quick context switch to alleviate the impact of memory latencies. Like ROCCC, the CHAT compiler is focused on generating hardware for highly specialized classes of input programs, such as sparse matrix multiplication. According to the authors, CHAT can translate only regular `for`-loops with a single index variable.

NYMBLE-MT and CHAT share the general idea that is beneficial to hide memory access latencies by switching execution to another ready thread. NYMBLE-MT, however, is capable of translating a much larger subset of C, demonstrated by its ability to create multi-threaded hardware accelerators for nine out of the 12 CHStone benchmarks [11].

In contrast, the LegUp developers pursue a different approach, more similar to software multi-threading [12]. LegUp accepts a parallel program that uses the *pthreads* and *OpenMPI* APIs, and generates a *dedicated hardware accelerator instance* for each (software) thread or for each parallel loop, respectively. This is fundamentally different from NYMBLE-MT, which aims to increase the utilization of a *single* accelerator instance by extending it for SMT multi-threaded execution

and allowing the processing of data from parallel threads in the *same* instance.

As an example for another completely different approach to multi-threaded accelerators, Convey Computer recently added support for a concept called *Hybrid Threading* (HT) to the tool chain for their FPGA-accelerated computing systems [13]. The HT flow accepts an idiomatic C++ description (basically an FSM, with each state representing a clock cycle, extended with message-based I/O) for efficiently describing computation, but without support for pointers or variable-bound/non-unit stride loops. These descriptions are then compiled into synthesizable HDL and linked to a vendor-supplied HW/SW framework that allows the starting of threads on the hardware accelerators and provides the context switching mechanism. Thread switching requires a single clock cycle and is used to effectively hide memory latencies. Despite being limited to an idiomatic programming style, the abstraction level of the HT C++ code is significantly higher than low-level HDL programming, with the actual multi-threading hardware being added automatically by the tools. The main difference between HT and NYMBLE-MT is, that the latter accepts true untimed programs, while HT relies on a manually scheduled/chained program with explicit message-based communication to host and memories.

III. Multi-Threaded Accelerators

Multi-threaded execution with re-ordering scheduling requires a controller capable of handling dynamic execution schedules in hardware. While it is possible to perform dynamic scheduling based on the readiness of *individual operators* [14], the resulting circuit complexity often leads to lower clock rates. Thus, we base our approach on introducing dynamicity at the granularity of entire *pipeline stages* (all operators within a stage must be ready for the stage to be considered ready). As shown in [8], this can implemented efficiently in the form of a Dynamic Stage Controller (DSC), which we use as base for the multi-threaded microarchitecture.

A. Execution Paradigm

The DSC is based on the principles of C-slow execution introduced by Leiserson et al. [15]. Data streams, externally interleaved/deinterleaved on a fixed round-robin (in-order) basis lead to an improvement in overall processing throughput. However, Leiserson's original approach is limited in that in applies only to *Fixed-Latency Operators* (FLO).

In the more powerful solution [8], all state in the accelerator is replicated N times for N threads, with the per-thread storage called the thread *context*.

In this manner, it is now possible to effectively handle VLOs, such as cached memory accesses. Once a thread has stalled in a stage on a cache miss, the next ready (un-stalled) thread in that stage can be scheduled. Also, in contrast to C-slow execution, we can now allow threads to *overtake* each other (re-order the schedule), basically dynamically changing Leiserson's interleaving scheme on-the-fly. This allows faster threads to continue without being stalled by other slower threads.

Stage	Stall
0	0
1	20
2	20
3	0
4	0
5	6434k
6	0
7	0

(a) Level 0

Stage	Stall
0	0
1	0
2	0
3	6433k
4	0
5	0
6	0

(b) Level 1

Stage	Stall
0	0
1	0
2	0
3	6428k
4	0
5	0
6	0

(c) Level 2

Stage	Stall
0	11700
1	1531116
2	1575000
3	1575000
4	1575000
5	0
6	0
7	0
8	0
9	450934
10	0
11	0

(d) Level 3

TABLE I
PER-STAGE STALL COUNTERS, WITH ONLY MANDATORY HTS ENABLED; BOXED=MANDATORY, GREY=OPTIONAL HTS

While this approach of replicating *all* state for N threads is effective, it is not efficient. A closer examination reveals that re-ordering of threads (to allow overtaking) can occur *only* in stages with VLOs, thus requiring per-thread context and a scheduling facility (both realized in the HTS) only in those stages. In all other stages, the conventional, non-duplicated pipeline state is used.

FLOs residing in (or spanning) pipeline stages together with VLOs must also be provided with context storage, as thread reordering may occur due to VLOs, and the active thread is tracked by the HTS only at the granularity of an entire pipeline stage. However, it suffices to place the context only at the end of such FLOs, as reordering cannot take place within them. The only issue to consider here is that the context must use queues sufficiently deep to buffer all of the data inside of a multi-cycle FLO, if the FLO cannot be completely stopped (e.g., by deasserting a Clock Enable signal), when the rest of the stage is stalled.

IV. Selective Placement of HTS Stages

As presented in [8], the approach discussed in the prior section already results in area efficient multi-threaded hardware-accelerators, where the throughput improvements significantly exceed the area growth due to more complex multi-threaded hardware.

But further area efficiency gains are possible. On closer examination, it turns out that the locations for HTS stages discussed in Section III-A actually fall into one of two categories: Those that are *mandatory* to achieve multi-threaded execution at all, and those that are *optional* and potentially just increase performance (by allowing thread reordering).

HTS stages are mandatory only to encapsulate entire loops, which are treated as VLOs from the perspective of the surrounding code. Without HTS support, only a single thread could enter a loop (while the rest would be blocked in a prior stage), which would prevent multi-threading especially in those parts of the program that could profit from them most. In all other places, HTS stages are just optional.

It is thus promising to explore if (and which) optional HTS stages could be removed from the accelerator with limited (or ideally even no) loss in performance. To this end, NYMBLE-MT was extended to insert per-thread performance counters into each DSC stage. These track the number of

cycles this stage would stall if only mandatory HTS stages were created. For stencil3d of the MachSuite [16] benchmark collection, this is shown in Table I along the HTS stage placement. The levels indicate the loop nesting levels, with 0 indicating the main function itself. Mandatory HTS stages are boxed, optional HTS stage have a grey background. As for all evaluations in this work, we generate an accelerator with four hardware threads. Also, we always target a Xilinx XC7VX690T device using Vivado 14.1 for logic synthesis and mapping.

Most of the activity occurs in the inner loop (at Level 3). Obviously, the optional HTS at Stage 4 would be useful (as its lack causes a large back-pressure of stalls, due to the inability to re-order threads in the *seven* memory read operators located at that stage). On the other hand, a HTS at Stage 9 would not be that useful (only a relatively small number of stalls occurs in the single write operator). The results of these studies have led to two proposed optimization heuristics.

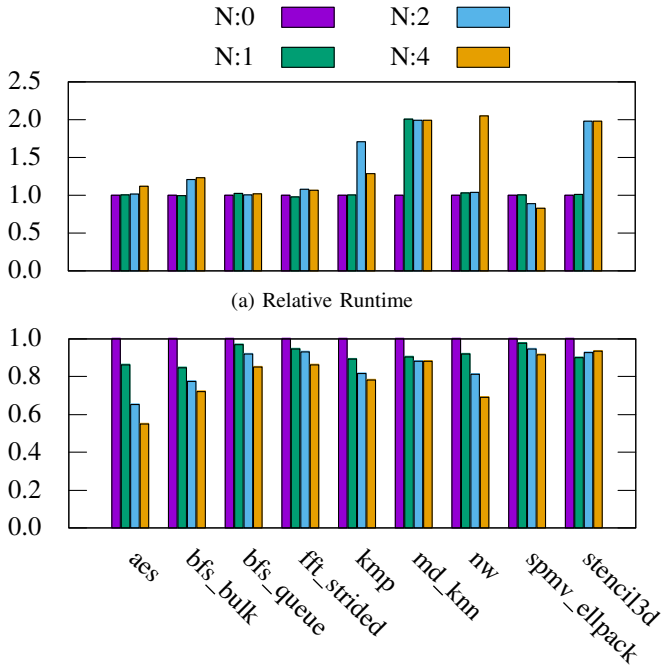
A. Backward HTS Deletion

When examining the profiling results for larger examples, it becomes clear that the lack of optional HTS stages in the later stages of the inner loops causes fewer stalls than having HTS capability missing from the earlier stages. This is already visible in the small stencil3d example of Table I, where the lack of HTS at Stage 4 is much more severe than on Stage 9 in the Level 3 loop.

This observation can be explained by considering the nature of datapath execution in the presence of VLOs. At the beginning of the loop (which itself is a VLO), all threads start at the same time, causing significant demand for shared ressources (such a memory accesses), and thus being most likely to stall. The variable-latency of the VLOs then causes a spreading-out-in-time of threads, as they get deeper in the pipeline (since the threads are often subject to different latencies). Thus, there is less potential for conflict among threads, and correspondingly less need for re-ordering by HTS.

A very simple heuristic can thus, for each loop level, attempt to omit the last N optional HTS stages, and only implement the earlier ones in each pipeline. The impact of this approach is shown in Figure 1, relative to an accelerator using *all* optional HTS stages. For each benchmark, each bar indicates the last $N = 0 \ldots 4$ HTS stages being dropped from each loop level. Figure 1.a gives the run-time in clock cycles (note: the clock frequency itself was not affected by the HTS removal) and .b the number of LUTs in the accelerator core (not including system interface logic).

The results are already promising: Even dropping only the last HTS stage (N:1) from a loop level can result in area savings of up to 20% (e.g., for aes), while maintaining the same performance of the fully-HTS-populated version. For many benchmarks, even the four last HTS stages can be dropped (N:4) without adversely affecting performance (e.g., also bfs_queue, spmv_ellpack), leading to area savings of up to 50% for aes. However, there is too much of a good thing: The performance of kmp begins to deteriorate if more



(a) Relative Runtime

(b) Relative #LUTs Kernel in Accelerator Core

Fig. 1. Greedily deleting the last N optional HTS stages

than one optional HTS stage is dropped, while md_knn cannot afford even the removal of a single level of HTS [1]. Obviously, a more targeted optimization strategy is required for consistent results.

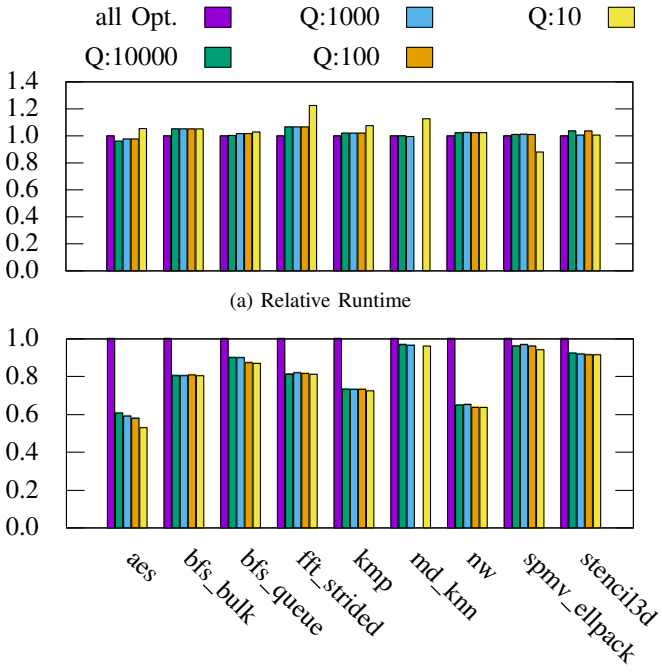
B. Profile-Guided HTS Insertion

HTS stages can be removed more selectively by taking the actual run-time behavior of the accelerator into account. This is achieved by relying on the performance counters described above to collect a stall profile when executing the accelerator (in simulation or hardware), with only mandatory HTS stages present, on representative input data.

The key idea here is to selectively insert only those optional HTS stages that are responsible for stalls that make up a significant fraction Q of the entire execution clock cycles. As the choice of Q is crucial for this heuristic, we will evaluate its performance over a wide range of values to examine the robustness of the algorithm. Note that this algorithm still relies on the already restricted places for HTS stages in general (see Section III-A). For stencil3d, this leads to the HTS-ineligible Stages $1 \ldots 3$ in Loop Level 3 (Table I.d) being disregarded for HTS insertion (despite their large stall counts), as the observed stalls are just the result of back-pressure that will be removed by a HTS inserted into Stage 4 (which, being optional, is actually eligible for HTS insertion).

As seen in Figure 2, the approach is robust with regard to the choice of Q. It already gives good results for $Q = 10,000$ (e.g., yielding 40% of LUT savings for aes), with only minor

[1]The odd effect of *increasing* performance when dropping more stages appears to be a side effect of also removing the priority-based scheduler in the HTS, which in itself might not be the best scheduling strategy for the specific benchmark, see Section V.



(a) Relative Runtime

(b) Relative #LUTs Kernel

Fig. 2. Profile-guided HTS insertion

area improvements achievable for $Q = 1,000 \ldots 100$. Only for $Q = 10$ becomes the insertion criterion too selective: A HTS would only be inserted into a stage with stalls exceeding more than 10% of the entire execution time. The benchmarks fft_strided and md_knn will be adversely affected by this, as too many useful HTS stages (but having smaller stall counts) would be omitted.

Compared to the Backwards Deletion heuristic, the results of the profile-guided approach are much more predictable (e.g., choosing $Q = 100$), while still realizing almost all the benefits (similar performance with less hardware area).

The key disadvantage of this approach is of course, that the HLS process now has to include the profiling (simulation or generation of actual hardware) of the accelerator. For the MachSuite benchmarks presented here, simulation takes between 4 and 14 minutes on current x86 compute servers.

V. Conclusion and Future Work

We have refined the concepts of SMT multi-threaded execution in HLS-generated accelerators by differentiating between mandatory and optional HTS stages. This distinction can be exploited to further reduce the area overhead of SMT execution, yielding reductions of up to 50% of LUTs and 50% of registers in SMT accelerators generated by HLS for the MachSuite benchmark collection.

We then presented two heuristics (one static, one dynamic) for maintaining SMT performance, while reducing the number of optional HTS stages. The later method is profile-guided, making it significantly more precise than the static one, but requires simulation time in the 10's of minutes during compilation, even for the relatively small benchmarks used in this study.

Future work will concentrate on improving the actual scheduling strategy beyond the simple priority based scheme used here. It is highly likely that different applications will require different strategies to achieve the best performance. To this end, we will evaluate both simple static methods (e.g., round-robin, pseudo-random) as well as truly dynamic approaches (tracking the past behavior of threads to make current scheduling decisions).

Acknowledgment

The authors would like to thank Xilinx, Inc. for supporting their work by hardware and software donations.

References

[1] Xilinx Inc., "Vivado Design Suite User Guide, UG902," 2012.
[2] Y Explorations Inc., "eXCite C to RTL Behavioral Synthesis."
[3] Synopsis Inc., "Synphony C Compiler User Guide," 2011.
[4] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "From Software to Accelerators with LegUp High-Level Synthesis," *ACM Transactions on Embedded Computing Systems*, vol. 13, no. 2, Sep. 2013.
[5] J. Villarreal, A. Park, W. Najjar, and R. Halstead, "Designing Modular Hardware Accelerators in C with ROCCC 2.0," in *2010 18th IEEE Annual Intl. Symp. on Field-Programmable Custom Computing Machines*, 2010.
[6] H. Gädke-Lütjens, "Dynamic Scheduling in High-Level Compilation for Adaptive Computers," Dissertation, TU Braunschweig, 2011.
[7] R. Nane, V.-M. Sima, B. Olivier, R. Meeuws, Y. Yankova, and K. Bertels, "DWARV 2.0: A CoSy-based C-to-VHDL hardware compiler," in *22nd Intl. Conf. on Field Programmable Logic and Applications (FPL)*, Aug. 2012.
[8] J. Huthmann, J. Oppermann, and A. Koch, "Automatic high-level synthesis of multi-threaded hardware accelerators," in *2014 24th Intl. Conf. on Field Programmable Logic and Applications (FPL)*, 2014.
[9] J. Huthmann, B. Liebig, J. Oppermann, and A. Koch, "Hardware/-software co-compilation with the Nymble system," in *2013 8th Intl. Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, Jul. 2013.
[10] R. Halstead and W. Najjar, "Compiled multithreaded data paths on FPGAs for dynamic workloads," *Compilers, Architecture and Synthesis* ..., 2013.
[11] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis," *Information and Media Technologies*, vol. 4, no. 4, 2009.
[12] J. Choi, S. Brown, and J. Anderson, "From Software Threads to Parallel Hardware in High-Level Synthesis for FPGAs," in *2013 Intl. Conf. on Field-Programmable Technology (FPT)*, Dec. 2013.
[13] Convey Computer Corp., "Hybrid Threading Reference Manual, Version 1.0," 2014.
[14] H. Gädke and A. Koch, "Accelerating speculative execution in high-level synthesis with cancel tokens," in *Reconfigurable Computing: Architectures, Tools and Applications*, 2008.
[15] C. Leiserson, F. Rose, and J. Sax, "Optimizing synchronous circuitry by retiming," in *Third Caltech Conf. On VLSI*, 1993.
[16] B. Reagen, R. Adolf, S. Y. Shao, G.-Y. Wei, and D. Brooks, "MachSuite: Benchmarks for Accelerator Design and Customized Architectures," in *IEEE Intl. Symp. on Workload Characterization (IISWC)*, 2014.