Automatic High-Level Synthesis of Multi-Threaded Hardware Accelerators

Hidden for review*, Hidden for review†, Hidden for review*
*Affiliation hidden for review
Email: Hidden for review
†Affiliation hidden for review
Email: Hidden for review

***Abstract*—With the growing maturity of high-level synthesis systems, the attractive capabilities of architectures relying on reconfigurable computing have become accessible to a wider range of developers. A promising next step of research is the efficient mapping of the high-level descriptions in languages such as C to more powerful micro-architecture templates. One scheme that can be explored here is that of Single Instruction Multi-Threaded (SIMT) execution, re-using the chip area dedicated to computation for different threads to hide memory latencies.**

We describe extending the hardware/software co-compiler Nymble to automatically generate such multi-threaded hardware accelerators. In contrast to prior work that simply duplicated complete compute units for each thread, Nymble-MT reuses the actual computation elements, and adds just the required data storage and context switching logic.

Evaluated using the CHStone benchmark suite and a sample configuration of four threads, the existing prototype can reach up to quadruple the throughput, but with a chip area just 5% larger than that of a single-threaded accelerator.

I. Introduction

With the basics of hardware synthesis from high-level languages now established both in industrial and academic tool sets, research now focuses on more specialized aspects of the flows. Current examples include studies on polyhedral optimization [1], or memory partitioning [2].

In this work, we extend our own prior research on Nymble [3], a hardware-software co-compilation system from C to shared-memory heterogeneous reconfigurable computers, with the capability to automatically synthesize *multi-threaded* execution units. For Nymble-MT, we will, e.g., selectively apply coarse-grained dynamic scheduling and per-thread context data storage. We aim to improve datapath utilization in the presence of variable memory latencies and dynamic loop-carried dependencies. Specific issues considered include the efficient modeling of memory dependencies in the synthesized controller as well as different thread scheduling options (in-order, reorderable) and their impact on chip areas and clock frequencies.

II. Related Work

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 [4], Y Explorations eXCite [5], and Synopsis Synphony C Compiler [6]. 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 [7], ROCCC [8], Comrade [9], and DWARV [10]. The topic of exploiting multi-threaded execution in the generated hardware is even more rarely addressed.

In [11], 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.

The Nymble-MT tool presented here 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 [12]. The remaining benchmarks `jpeg` and `aes` contain non-inlineable function calls, which could be handled using the software-service call feature of Nymble, but that has not been enhanced yet for multi-threading in Nymble-MT. For the last missing benchmark `motion`, the CHStone-supplied test harness passes parameters in a way that is incompatible with our current simulation framework.

In contrast, the LegUp developers pursue a different approach, more similar to software multi-threading [13]. LegUp accepts a parallel program that uses the *pthreads* and *Open-MPI* 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 multi-threaded execution and allowing the processing of data from parallel threads.

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 [14]. 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

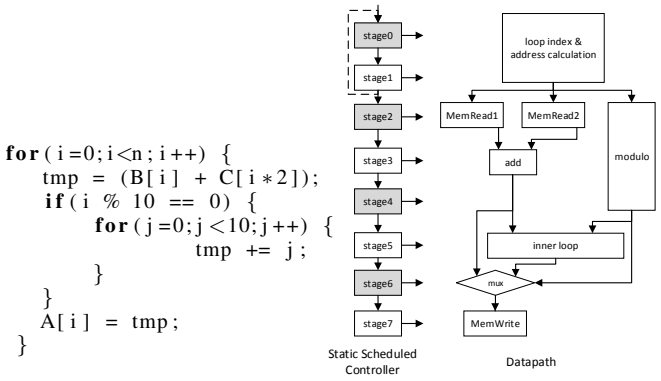

Fig. 1. Example for a statically scheduled datapath with II=2. Shaded stages are executed simultaneously using pipelining

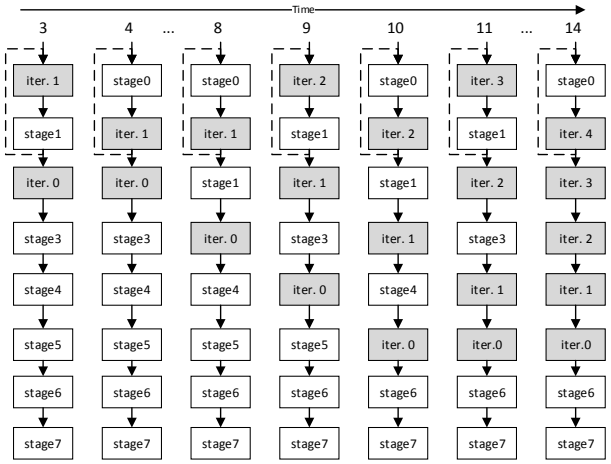

Fig. 2. Dynamic stage controller executing the example in Fig. 1

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-Threading

The work presented here builds on the Hardware/Software Co-Compiler Nymble [3]. Nymble uses mainly static scheduling and assigns each operator a specific clock-cycle for execution (called *stage*). However, it does support variable-latency operations [15] or hardware-to-software calls [16] by stalling the entire datapath, even though some datapath regions might be completely independent of the variable-latency operations. Also, while Nymble pipelines the execution, the utilization of the data paths is highly dependent on the loop-carried dependencies (LCD), which often limit the initiation interval (II) to values greater than one. This section describes the different techniques we explored to improve the utilization of Nymble-created hardware accelerators.

Figure 1 shows an example of a statically scheduled datapath. For clarity, we omitted the details of the inner loop, the loop counter and address calculations. The outer loop shown here has II=2, since an LCD exists on incrementing the counter variable (indicated as a dashed data back-edge in the figure). The computation is organized as stages holding one or more hardware operators, with the controller being shown in the center and the associated datapath at the right side of the figure.

A. Dynamic Stage Controller

An obvious approach to increase the utilization is to limit the effects of stalling to just the dependent regions of the datapaths. At the extreme end, this can be done at the level of individual operators [17]. However, this very fine granularity carries at a high area overhead and can lead to slower clock rates due to significantly increased wiring requirements of per-operator token networks. As a compromise, we propose the synthesis of datapaths that can be stalled at the granularity of individual stages, allowing prior and successive stages independent of the stalled stages to continue. This functionality is provided by a Dynamic Stage Controller (DSC), which is automatically generated for each high-level input code.

Figure 2 demonstrates how the proposed model of execution applies to the Example in Figure 1. At Time 3, Iteration 0 stalls as one of the memory reads in Stage 2 is assumed to suffer a cache miss and thus cannot immediately provide a value. Using a DSC, Iteration 1 is allowed to proceed at Time 4 from Stage 0 to Stage 1 despite the stall. However, since Stage 2 is still occupied by Iteration 0, it cannot continue, until the read is satisfied at Time 8. Then, Iteration 0 vacates Stage 2, allowing Iteration 1 to complete and move on at Time 9. That, in turn allows the initiation of Iteration 2 in Stage 0. At Time 10, Iteration 0 has reached the inner loop, which is also treated as a variable-latency operator from the perspective of the outer loop. It appears to be stalled there. However, independent stages of the datapath continue executing. Assuming cache hits in the outer loop while the inner loop still executes, this eventually allows Iterations 1 to 4 to catch proceed until Iteration 1 reaches the inner loop (still occupied by Iteration 0). Execution continues only after the inner loop has completed for outer loop Iteration 0.

The DSC implementation is a streamlined version of the more powerful controller we proposed in [18] for data value speculation. Figure 3 shows the basic structure: The completion state is tracked per stage, not per operator as in [17]. This can be seen in Stage 2: Only if both reads complete,

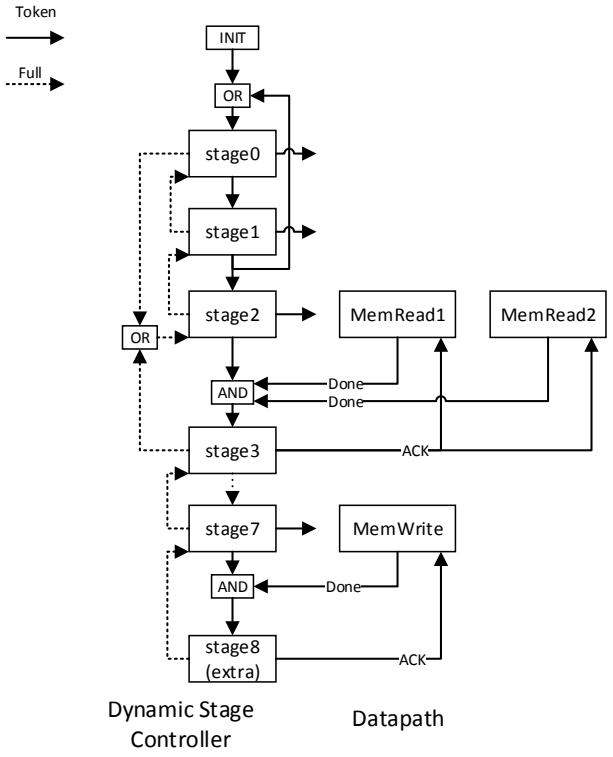

Fig. 3. Dynamic Stage Controller

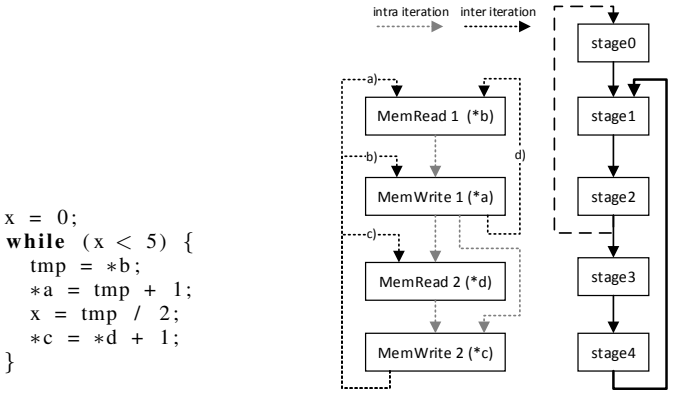

Fig. 4. Handling memory dependencies (II=3)

execution is allowed to proceed. Back-pressure (indicating a still-busy successor stage) is handled using Full signals, which can cause the data in the predecessor stage to be held there until the successor stage becomes available.

However, an additional issue needs to be addressed: In [18], memory dependencies were optimistically resolved using data value speculation, replaying computations executing on misspeculated values. Since the DSC aims for a more area efficient approach, omitting the speculation and replay logic, memory dependencies need to be resolved conservatively in the DSC. In general, any kind of dynamic execution needs to explicitly track memory dependencies, e.g., using special activation tokens [9]. However, at the stage-based granularity used in DCS, explicit dependencies can often be omitted (relying on the staged execution order), or be folded onto other dependencies. Both ways reduce the number of dependencies that need to be explicitly tracked using memory tokens.

Figure 4 illustrates this on a simple example with II=3: For the code fragment given on the left side, the potential memory dependencies shown in the center exist (distinguishing between intra- and inter-iteration dependencies). Intra-iteration dependencies do not need to be tracked explicitly, as assigning each memory operation to a stage (Stage 1 ...4) in program order ensures that these dependencies will be satisfied. On the other hand, without explicit tracking, the DSC cannot ensure that MemWrite2 is always executed *prior* to MemRead1 (Inter-iteration dependency *a*), MemWrite1 (Inter-iteration dependency *b*), and MemRead2 (Inter-iteration dependency *c*). Unless it can be proven (e.g., using alias analysis), that these potential dependencies do not actually exist, this execution order must be guaranteed using explicit tracking in the DSC.

In a purely operator-based dynamic execution model, MemRead1 would wait for two individual tokens from its inter-iteration predecessors MemWrite1 and MemWrite2, requiring logic and wiring area on the device. However, the tracking effort can be reduced by exploiting the stage-based execution, leading to a simplified controller as shown at the right side of Figure 4: Dependency *(d)*, execute MemWrite1 prior to MemRead1 of the next iteration, is *already* enforced by the LCD data edge (Stage 2, Stage 1) that ensures the correct update of the loop decision variable x (which is the root cause for II=3). Memory edges *(b)* and *(c)* can be folded onto memory edge *(a)*, which, together with the natural stage order present in the DSC, ensures that MemWrite1 and MemRead2 will be executed after MemWrite2. Thus, in this example, only that last memory dependency needs to be explicitly tracked, leading to an edge (Stage 4, Stage 1) in the DSC.

In hardware, the DSC as shown in Figure 3 is realized as a shift-register, with a flip-flop for each stage to indicate an active stage. Simple logic provides the basic transition rules: advance to the next stage only if *all* variable-latency operators in a stage have completed, remain in this stage if successor stage is still full (any operators are busy). This basic scheme is extended with the additional memory dependency tracking logic that handles the per-stage dependency tokens. These are generated when the predecessor stage of a stage holding one or more operations acting as dependency sources has completed. For loop entry, the DSC has a special INIT state that is initialized to '1' to start execution.

B. Multithreaded Execution

While the DSC can already improve utilization of the datapath hardware area, further gains are possible. As we will demonstrate, even for FPGA-based hardware accelerators, it can be efficient (in terms of area required / throughput gained) to exploit multi-threaded execution (sometimes also called SIMT or SMT).

In its simplest form, for improved utilization, we could process N independent data streams in the datapath, where $N = II$ to improve throughput. The data streams would be

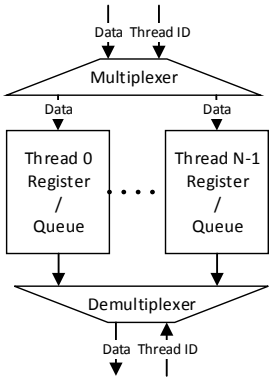

Fig. 5. Thread context storage

externally interleaved/deinterleaved on a fixed round-robin (in-order) basis, using the principles of C-slow execution (with $C = N$) introduced by Leiserson et al. [19]. However, the original approach is limited in that in applies only to constant-latency operators.

In a more powerful solution, all state in the accelerator is replicated N times for N threads, with the per-thread storage called the thread *context*. Figure 5 show this duplication and how all context accesses (reads and writes) are qualified by the ID of the current thread (TID). This is easily added to the DSC controller: Instead of just using a '1' to indicate that a stage is active, the controller now stores the TID of the thread active in that stage.

With thread context, we can achieve two goals: First, it is now possible to handle variable-latency operators such as cached memory accesses. Once a thread has stalled in a stage on a cache miss, we can schedule the next ready (un-stalled) thread in that stage. Second, in contrast to C-slow execution, we can now allow threads to *overtake* each other, basically dynamically changing Leiserson's interleaving scheme on-the-fly. To continue the example, if a second thread had a cache hit in the stage that stalled the prior thread, it could continue execution past the earlier thread. For each variable-latency operation in a stage, an external arbiter keeps track of the per-thread data availability, and selects one of those threads that has all required data present in the datapath (e.g., a cache hit, or a variable-latency computation such as an inner loop completed) to proceed to the next stage.

However, similar to the folding of memory dependence edges in Section III-A, we can also reduce the extent of state duplication for context *below* the factor N required by the brute-force approach. Overtaking of threads can occur only in stages with variable-latency operations, thus we require context only in those stages. In all other stages, we just use the conventional, non-duplicated pipeline state. This is shown in Figure 6 for the Example of Figure 1: Only the stages with bold borders hold context.

Stage 0, even though it does not hold variable-latency operations, also needs context in order to avoid deadlocks: One or more threads could compete for Stage 0 via the loop

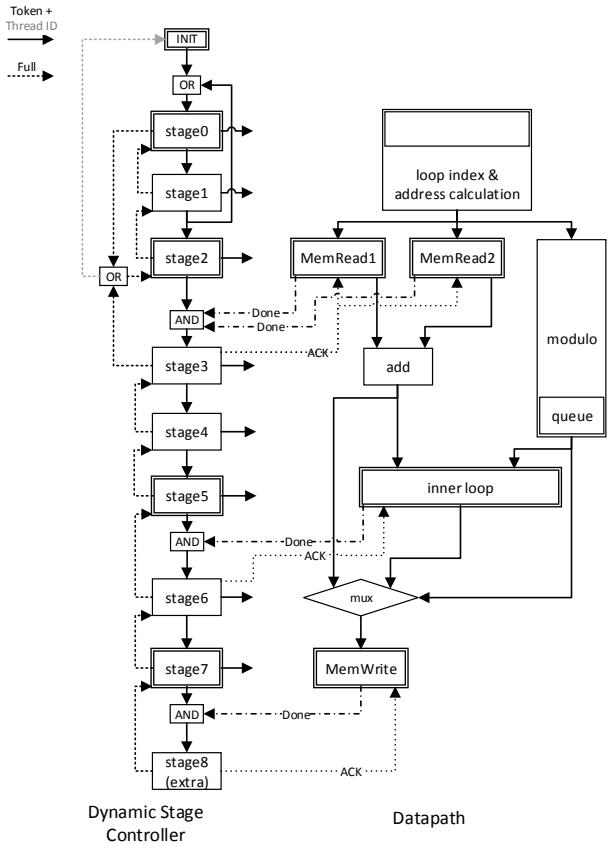

Fig. 6. Example datapath with II=2, bold operations contain thread context (duplicated state)

back-edge. If none of them could proceed due to Stage 0 being exclusively busy with another thread, the resulting back-pressure along the back edge could lead to busy stages all the way back up to Stage 1 (successor of Stage 0), which would prevent Stage 0 itself from ever completing and thus deadlock the entire datapath. Since both memory accesses and variable-latency operators will eventually complete, they do not cause deadlocks by themselves.

Fixed-latency multi-cycle operations that span stages holding variable-latency operators must also be provided with context, as thread reordering may occur due to the other operators in a stage, and the active thread is tracked only for an entire stage. However, it suffices to place the context only at the end of such operations, as reordering cannot take place within them. In this manner (context storage at the outputs), non-threaded third-party IP blocks can easily be integrated into compiled threaded datapaths. The only issue to consider here is that the context must use queues deep enough to buffer all of the data inside of the multi-cycle operation, if the module cannot be completely stopped (e.g., by deasserting a clock enable) when a stage has to be stalled. E.g., for a 32-cycle modulo operator, the context consists of a 32-element queue for each thread.

Figure 7 shows two possible scenarios (named I and II) for the reordering of two threads. At Time 3, Iteration 0 of thread

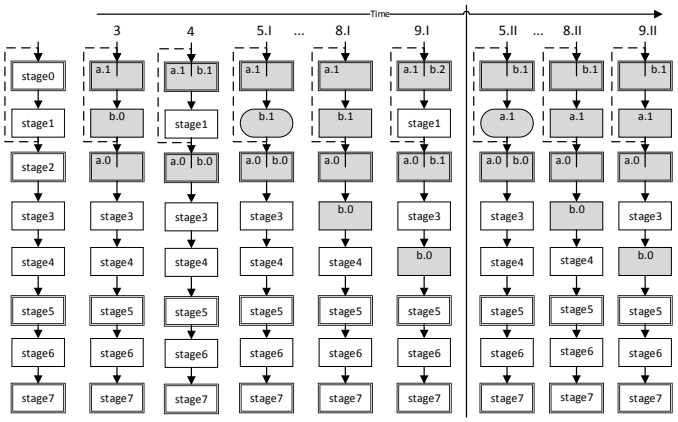

Fig. 7. Thread reordering example, for scenarios I and II. Labels have the form [thread id . iteration].

a, written as a.0, reaches the memory read at Stage 2 and is assumed to stall due to cache miss. b.0, which was started one cycle after a, has advanced to Stage 2 then. Since the example assumes II=2, a.1 is also started at Time 3. Since we are still relying on the DSC, the stall in Stage 2 does not hinder other stages. Thus at Time 4, b.0 is allowed to advance into Stage 2, which already holds the stalled a.0. Assuming that b.0 also experiences a cache-miss, which is resolved by Time 7, b.0 would then be allowed to advance to Stage 3 at Time 8, overtaking a.0 which is still stalled in Stage 2.

Meanwhile, we have to pick a thread to advance into Stage 1. As Stage 1 is not a variable-latency operation, it does not have context storage and can thus hold only a single thread. Two options exist: In Scenario I, at Time 4 b.1 is selected, which could then proceed to Stage 1 (Time 5) and Stage 2 (which is assumed to still hold a stalled a.0) at Time 9. In Scenario II, a.1 is picked instead at Time 4. While a.1 can advance to Stage 1 at Time 5, further advancement is not possible as Thread a already occupies Stage 2 in the form of a.0. Even worse, since Stage 1 has no context, b.1 also has to remain in Stage 0. This will change only once the memory request of a.0 in Stage 2 has actually been satisfied, then a.0 moves on and a.1 can enter Stage 2 (not shown in the figure).

The current implementation generates a simple priority encoder for thread scheduling, always picking the thread with the lowest thread ID. Future work could improve this, e.g., picking threads which experienced cache hits in the past (to exploit locality).

Another option, which we will evaluate in Section IV, will be the use of *queues* instead of the simple duplicated registers for storing context. In the example of Figure 7, such a queue would allow a.1 to enter Stage 2 at Time 6 in Scenario II, even though a.0 is already present (earlier in the queue). Note that this use of queues is different from the one outlined earlier (buffering pipelined results of unstoppable IP blocks which span stalled stages).

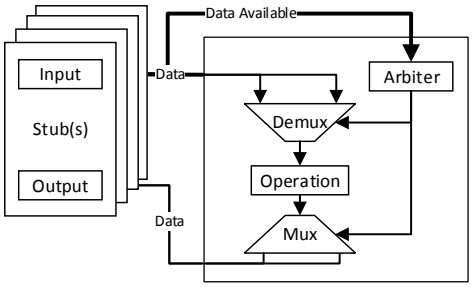

Fig. 8. Multiplexer for variable latency operations

C. Multiplexing Memory Ports

Nymble-MT tries to follow a fully spatial execution model when building its datapaths. However, that becomes infeasible when accesses to external memory are considered. Here, resources (e.g., memory controller, memory bus(es) etc.) have to be shared between the many memory operations often present in real applications. In the purely-statically scheduled Nymble microarchitecture [3], this was achieved by imposing an upper limit on the number of parallel memory operations per stage for scheduling.

In Nymble-MT, however, the stage-based *dynamic* scheduling makes such *static* limits infeasible. Instead, shared resources such as memory ports must be arbitrated dynamically. This is done in a generic way, which could also be used to share other large resources such as FPUs.

As shown in Figure 8, each of the resources to be shared is wrapped in stubs that just provide input/output registers for an operation. An arbiter observes the presence of data on the input registers. If a stub has valid data in all of its input registers, it is eligible for access to the shared resource. The arbiter currently uses a simple static priority scheme to grant access if multiple stubs are eligible for execution (this could be refined in future work, especially in conjuction with better thread scheduling). The input data of the selected stub is then forwarded to the actual shared resource, and the result(s) then passed back to the stub output registers.

IV. Experimental Evaluation

We evaluate our approach by exploring a number of different micro-architecture options on the CHStone benchmark suite. Like its predecessor, Nymble-MT aims for fined-grained hardware/software co-execution. This is reflected in our choice of target platform: We employ a Xilinx ML507 board (Virtex 5 FX-based), using the hardware and software environment described in [20] to achieve high-throughput low-latency access to shared memory between the accelerator(s) and the general-purpose PowerPC 440 processor. As the XC5VFX70 device on the actual board is too small to hold the complete system-on-chip (processor buses, memory controller, network interface, etc.) for the larger CHStone examples, we also employ a simulated version of the board that virtually substitutes the larger XC5VFX200 device into the same architecture. For our measurements, we performed post-layout simulations,

	#Clock Cycles						Clock Frequency [MHz]					
	Single Threaded		Four Threaded				Single Threaded	Four Threaded				
Benchmark	static	DSC	RO=0 Q=0	RO=0 Q=1	RO=1 Q=0	RO=1 Q=1	static	RO=0 Q=0	RO=0 Q=1	RO=1 Q=0	RO=1 Q=1	Best f of 4 threads
adpcm	66659	81768	85448	85447	83817	83817	72	*	*	*	*	*
blowfish	1467210	1153336	2094483	2092148	1895012	1891892	87	84	93	82	72	93
dfadd	11610	14268	36918	36918	36918	36918	100	93	95	93	93	95
dfadd_mod	5357	5312	14578	14578	11611	11611	101	89	80	78	73	89
dfdiv	9398	10893	18930	18930	18884	18930	95	90	94	93	89	94
dfmul	6078	6314	16054	16054	16054	16054	100	92	93	95	91	95
dfmul_mod	1883	1714	3459	3429	2427	2632	92	85	88	75	74	88
dfsin	201338	222435	418346	418346	413661	413661	79	40	49	*	*	49
example	30707	24035	82572	81724	56331	56399	93	100	94	86	94	100
example -O0	33072	19442	72098	70009	60695	53372	100	91	91	93	84	93
gsm	27082	19543	51173	33742	28458	20221	76	82	75	69	49	82
mips	37837	28327	29536	29568	28871	28924	100	85	84	75	75	85
sha	683834	771098	999806	999806	773284	773284	100	79	94	77	94	94

TABLE I
EXECUTION CYCLES AND CLOCK FREQUENCIES

including cycle-accurate models for memories and caches. All Nymble-family compilers use LLVM as front-end and for machine-independent optimization. The specific version employed here is LLVM 3.3. Furthermore, we relied on Mentor Precision 2013b5 for logic synthesis, and Xilinx ISE 14.6 for physical mapping.

As benchmarks, we used CHstone and one additional test case: example is the program shown in Figure 1. For the examples dfadd, dfmul and example, we explore additional implementation alternatives discussed in the next subsection.

The column static refers to a purely statically scheduled data path, DSC uses the Dynamic Stage Controller, and both are operating in single-threaded execution mode. The remaining cases compile to accelerators processing four threads. For them, we explore the impact of allowing thread-reordering (RO=1) and using 16-entry queues for context instead of just registers (Q=1). As DSC is just a RO=0, Q=0 implementation executing a single thread, we have omitted its area numbers. Alternatives marked with * in the table did not map successfully to the XC5VFX200[1]. Also, in some cases, the DSC has a higher cycle count than the static datapath, since the latter is able to optimistically start a new iteration (and cancel too-aggressive results afterward). The DSC always has to completely evaluate the control condition of the loop before starting a new iteration.

In the multi-threaded cases, time is measured from the start of the earliest thread to the completion of the last thread, with a thread being started every 10 clock cycles. All time measurements include the execution time of the software thread-management API (memory-mapped communication) and the hardware cycles for invalidating (at start) and flushing (at end) the distributed per-thread caches.

A. General Discussion

Multithreading carries both an delay and area overhead (shown in Tables I and II) . This is reflected in the increased number of logic resources and lower achievable clock frequencies. However, when the overall *efficiencies* are examined, multi-threading can be beneficial. We will examine both time and area efficiencies, which for the four-threaded case are:

$$e_t = 4\ t_{\text{single}}/t_{\text{multi}} \qquad e_a = 4\ a_{\text{single}}/a_{\text{multi}}$$

Here, t indicates the wallclock execution time and a the accelerator area in slices, for both single-threading and multi-threading variants.

For $e_t > 1$, executing four threads simultaneously is faster by a factor of e_t than executing a single thread sequentially four times. Analogously, if $e_a > 1$, a four-threaded accelerator takes just $1/e_a$ of the area of four individual instances of a single-threaded accelerator. For all mappable benchmarks, both e_t and ea always exceed 1.0, in some cases even achieving close to perfect scaling effiency ($e_a = 3.85$ for dfdiv, $e_t = 3.84$ for mips, with the value 4.0 being perfect). If the optimization goal is throughput, and the often longer latency of the multi-threaded accelerator is not an issue, high-level synthesis using Nymble-MT is always worthwhile here.

dfadd_mod and dfmul_mod are modified versions of the corresponding CHStone benchmarks that execute the entire inner loop, instead of just the actual dfadd and dfmul kernel, on the accelerator. Avoiding hardware/software execution switches in this manner leads to significant speed-ups. However, even then, total performance gains for these benchmarks is somewhat limited since they execute only relatively few simple integer computations compared to the overhead of context storage and memory system.

Interestingly, in the dfdiv case, the number of registers of the multi-threaded implementation was actually less than in the statically-scheduled variant. The latter had to be constrained to issue a maximum number of one memory access per cycle, leading to long schedules with a large number of pipeline registers. The multiplexing and dynamic scheduling allowed to place an unlimited number of memory accesses into a single stage, thus leading to shorter schedules and fewer pipeline registers.

[1]We are currently performing additional mapping runs with different options, these results will be included in the final paper.

	#Slices						
	Single Threaded	Four Threaded					
Benchmark	static	RO=0 Q=0	RO=0 Q=1	RO=1 Q=0	RO=1 Q=1	Size of fastest wallclock 4-thr.	e_a
adpcm	28247	*	*	*	*	*	*
blowfish	10341	17503	18265	22828	23473	18265	2.26
dfadd	8426	12547	12612	13892	14010	12612	2.67
dfadd_mod	7986	16092	17106	20019	25742	20019	1.60
dfdiv	18911	19476	19669	19589	19876	19669	3.85
dfmul	7408	10949	10883	11073	11338	11073	2.68
dfmul_mod	7951	14501	15358	18117	22310	18117	1.76
dfsin	28885	30716	30645	*	*	30645	3.77
example	5553	8440	8557	8712	9180	9180	2.42
example -O0	5638	8906	9070	8980	9602	9602	2.35
gsm	21408	22600	24018	24279	28918	24279	3.53
mips	8745	13391	12908	16495	16495	13391	2.61
sha	18359	26534	26715	28453	29044	29044	2.53

TABLE II
AREA REQUIREMENTS AND AREA EFFICIENCY e_a

B. Impact of Queues and Thread Reordering

For the discussion of these features, consider the benchmark example, corresponding to Figure 1. All of the memory accesses (arrays A, B, and C) are assumed to be independent. The discussion of Figure 2 and Figure 7 already sketched the potential effects of allowing thread reordering and using queues for context storage. This can now be shown in practice when considering the cycle counts of the different implementations of the example test case. Furthermore, we can use this benchmark to illustrate possible interference between machine-independent optimization in LLVM and the actual high-level synthesis by Nymble-MT.

For beginning the discussion, assume that the hardware generated exactly reflects the code structure as actually written in Figure 1 (two nested loops). This can be achieved in practice by disabling machine-independent optimizations in LLVM (similar to -O0 in other compilers). Nymble-MT then indeed synthesizes a datapath with II=2. With that short an II, queues are quite useful to reduce the back pressure, allowing a thread to advance into a stage even though it is already occupied by a previous iteration of the same thread (Scenario II, as described in Section III-B and Figure 7 for a.0 and a.1). Execution for example -O0 drops from 60,695 cycles without queues to 53,372 cycles with 16-entry queues and reordering. Without reordering, throughput is so limited that back pressure does not build in the datapath, in which case queues do not gain much performance (72,098 vs. 70,009 cyles). In general, for short IIs, reordering works best with queues enabled.

For the same reason, datapaths with longer IIs do not profit much from enabled queues. Compiling example with -O3 (the default in our flow) leads to the inner loop being completely removed (due to LLVM -O3 pass indvars statically determining that tmp will always incremented by 45, and then removing the useless loop by the loop-deletion pass). However, subsequent -O3 optimizations instcombine and loop-rotate require the introduction of additional control logic. From the hardware perspective, these transformations actually lead to II=6, which is sufficiently long to avoid building intra-thread back pressure and thus does not profit from queues (slowdown from 56,331 to 56,399 cycles). A better choice of machine-independent optimizations and an algorithm for selectively placing reordering stages and queues only at profitable stages appear to be highly profitable for future work.

C. Implementation Limitations

At this stage, Nymble-MT is a proof-of-concept prototype and still suffers from a number of implementation weaknesses.

For shorter benchmarks (e.g., dfdiv and dfmul), the co-execution overhead of the software API managing the four hardware threads becomes significant (roughly 75% of execution cycles). Here, a tighter interface than that achievable on the ML507 (non-real time Linux communicating via PLB with the accelerator and using IRQ-based signaling) would be preferable.

Some of the loss in clock frequency is currently due to operator chaining being performed in the compile-flow *before* context storage is inserted. By making chaining context-aware, better results could be achieved.

Area could be reduced further by globally multiplexing memory accesses *across* loop boundaries. Since we use distributed caches for memory accesses, reducing the number of cache ports by multiplexing across sequentially executed loops could significantly reduce area and also improve clock rates (due to reduced routing pressure at the cache-memory controller interface).

V. CONCLUSION AND FUTURE WORK

We have shown that a combination of different techniques such as dynamic stage-based scheduling, state duplication, queue insertion, and thread-reordering allow the automatic high-level synthesis of multi-threaded hardware accelerators which have better time and area efficiencies than temporally reused or spatially replicated single-threaded microarchitectures.

A number of areas for future work have already been identified. These include better thread scheduling techniques prioritizing non-stalled threads, more intelligent multiplexing of memory ports exploiting temporal and spatial locality, and context-insertion aware chaining to achieve higher clock

	Wallclock Execution Time [ms]								
	Single Threaded			Four Threaded					
Benchmark	static	DSC	Best t of static or DSC	RO=0 Q=0	RO=0 Q=1	RO=1 Q=0	RO=1 Q=1	Best wallclock	e_t
adpcm	0.926	*	*	*	*	*	*	*	*
blowfish	16.864	12.401	12.401	24.934	22.496	23.110	26.276	22.496	2.21
dfadd	0.116	0.150	0.116	0.397	0.389	0.397	0.397	0.389	1.20
dfadd_mod	0.053	0.060	0.053	0.164	0.182	0.149	0.159	0.149	1.43
dfdiv	0.099	0.116	0.099	0.210	0.201	0.203	0.213	0.201	1.96
dfmul	0.061	0.066	0.061	0.175	0.173	0.169	0.176	0.169	1.44
dfmul_mod	0.020	0.019	0.019	0.041	0.039	0.032	0.036	0.032	2.41
dfsin	2.549	4.539	2.549	10.459	8.538	*	*	8.538	1.19
example	0.330	0.240	0.240	0.826	0.869	0.655	0.600	0.600	1.60
example -O0	0.331	0.209	0.209	0.792	0.769	0.653	0.635	0.635	1.32
gsm	0.356	0.238	0.238	0.624	0.450	0.412	0.413	0.412	2.31
mips	0.378	0.333	0.333	0.347	0.352	0.385	0.386	0.347	3.84
sha	6.838	8.203	6.838	12.656	10.636	10.043	8.226	8.226	3.33

TABLE III
WALLCLOCK EXECUTION TIME AND TIME EFFICIENCY e_t

frequencies. Hardware/software co-execution latencies could be reduced by porting the required infrastructure to a platform with tighter integration between software-programmable general-purpose processor and hardware accelerators, such as the Xilinx Zynq family. The latter effort is already under way.

ACKNOWLEDGMENT

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

REFERENCES

[1] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong, "Polyhedral-based data reuse optimization for configurable computing," in *Proc. of the ACM/SIGDA Intl. Symp. on Field Programmable Gate Arrays*, 2013.

[2] Y. Wang, P. Li, and J. Cong, "Theory and algorithm for generalized memory partitioning in high-level synthesis," in *Proc. of the 2014 ACM/SIGDA Intl. Symp. on Field-programmable Gate Arrays*, 2014.

[3] XXX, B. Liebig, XXX, and XXX, "Hardware/software co-compilation with the Nymble system," in *2013 8th Intl. Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, Jul. 2013.

[4] Xilinx Inc., "Vivado Design Suite User Guide, UG902," 2012.

[5] Y Explorations Inc., "eXCite C to RTL Behavioral Synthesis."

[6] Synopsis Inc., "Synphony C Compiler User Guide," 2011.

[7] 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.

[8] 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.

[9] H. Gädke-Lütjens, "Dynamic Scheduling in High-Level Compilation for Adaptive Computers," Dissertation, TU Braunschweig, 2011.

[10] 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.

[11] R. Halstead and W. Najjar, "Compiled multithreaded data paths on FPGAs for dynamic workloads," *Compilers, Architecture and Synthesis* . . . , 2013.

[12] 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.

[13] 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.

[14] Convey Computer Corp., "Hybrid Threading Reference Manual, Version 1.0," 2014.

[15] H. Gädke-Lütjens, B. Thielmann, and XXX, "A flexible compute and memory infrastructure for high-level language to hardware compilation," in *Field Programmable Logic and Applications (FPL), 2010 Intl. Conf. on*, 2010.

[16] H. Lange and XXX, "An execution model for hardware/software compilation and its system-level realization," in *Field Programmable Logic and Applications, 2007. FPL 2007. Intl. Conf. on*, 2007.

[17] H. Gädke and XXX, "Accelerating speculative execution in high-level synthesis with cancel tokens," in *Reconfigurable Computing: Architectures, Tools and Applications*, 2008.

[18] B. Thielmann, XXX, and XXX, "Evaluation of speculative execution techniques for high-level language to hardware compilation," *6th Intl. Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC)*, Jun. 2011.

[19] C. Leiserson, F. Rose, and J. Sax, "Optimizing synchronous circuitry by retiming," in *Third Caltech Conf. On VLSI*, 1993.

[20] H. Lange and XXX, "Architectures and execution models for hardware/-software compilation and their system-level realization," *IEEE Trans. Comput.*, vol. 59, no. 10, Oct. 2010.