A FLEXIBLE COMPUTE AND MEMORY INFRASTRUCTURE FOR HIGH-LEVEL LANGUAGE TO HARDWARE COMPILATION

Hagen Gädke-Lütjens, Benjamin Thielmann

Integrated Circuit Design (E.I.S.)
Technische Universität Braunschweig
email: {gaedke|thielmann}@eis.cs.tu-bs.de

Andreas Koch

Embedded Systems and Applications Group
Technische Universität Darmstadt
email: koch@esa.cs.tu-darmstadt.de

Revised version (Feb. 2011)

- Section 7.D / Table 2

ABSTRACT

We present a low-level infrastructure for use by high-level language to hardware compiler back-ends. It consists of the highly parameterizable, technology-independent module library Modlib and the LMEM framework for localizing variables in fast on-chip memories. Modlib not only supports all high-level language operators (including memory accesses), but also provides a wide spectrum of usage modes: covering static and dynamic scheduling, speculative predicated execution, pipeline balancing, and explicit canceling of mis-speculated computations. We examine the performance of the infrastructure for a number of automatically compiled kernels, including an MD5 kernel that significantly profits from using LMEM.

1. INTRODUCTION

Adaptive computing systems (ACS) combine a software-programmable central processing unit (CPU) and a reconfigurable compute unit (RCU) to run compute-intense kernels as hardware implementations on the RCU.

However, actual application development for an ACS requires both hardware (HW) and software (SW) design skills and, for the RCU, is often performed in dedicated HW description languages (HDL) at a low level. We have been working on the COMRADE flow aiming to compile conventional ANSI C programs into hybrid HW/SW solutions [15].

When creating HW from C, a central issue is the mapping of language operators to actual HW units. Current solutions range from a simple direct 1:1 mapping to HDL operators to intermediate parametrized module generator libraries. Furthermore, beyond the C operator semantics, the mapping to HW units also has to deal with issues such as time (combinatorial delay, fixed or variable sequential latency) and control interfaces for scheduling (e.g., static, dynamic; predicated; speculative; explicit cancellation of mis-speculated operations [5] [8], etc.). Although not immediately obvious, memory accesses (pointer and array operators) also need to be covered.

In this paper, we present two low-level building blocks of our approach: *Modlib*, a mostly technology-independent general-purpose module library covering all C operators. *LMEM* is an infrastructure supporting fast localized on-chip memories providing the compiled RCUs with high-bandwidth low-latency data access.

The specific contributions of this work are the highly flexible Modlib library and the light-weight memory-localization infrastructure LMEM. Modlib remains competitive with more limited prior approaches such as GLACE [25], even when generating dynamically scheduled modules. Modlib can insert queues to achieve looser coupling within the data paths, a feature will be examined in conjunction with LMEM for accelerating the MD5 algorithm. The results compare favorably with some of the best manually optimized designs.

2. RELATED WORK

High-level language to HW compilers have widely varying feature sets. In many cases, they concentrate on the efficient translation of just a limited subset of the input language's feature set. For C, this might include only non-nested loops with fixed bounds and step sizes, or no irregular control flow (e.g., `break`, `continue`) in loops, only streaming access to memory, etc. Such compilers generally make do with a simple static scheduling of operators and often do not consider variable-latency operations such as cached memory accesses. Beyond the scheduling, the compilers also differ in the kind of operators and data types supported.

Some examples of this more limited approach (which, despite its restrictions, can still yield sizable productivity gains over pure RTL HDL designs) are: Celoxica Handel-C [23], ROCCC [6], and SPARK [7], which do not support division at all. Synfora PICO Express [31], which does not support floating point. CompiLogic C2Verilog [27] (acquired by Synopsys) implements division and modulo only as single-cycle combinatorial logic, possibly leading to low operating frequencies. Floating point operators, while present, have low throughput. UCLA xPilot [3] compiles the C division operator directly into a HDL division operator, also leading to high-delay combinatorial HW.

Compilers using dynamic scheduling can better handle irregular control flow (e.g., variable loop iteration intervals) and variable latency operators (e.g., stall just operators data-dependent on a cache-missed memory load instead of the entire RCU). For the module library, this generally implies the need for more sophisticated hand-shake protocols indicating data availability and operator readiness. Examples of compilers creating dynamically scheduled HW are CASH [2], Molen [26], Xilinx CHiMPS [28], and COMRADE [15] [5] [8]. While all of these tools use dynamic scheduling, they differ widely in their actual HW generation: The Verilog back-end of CASH does not support pipelined and floating point operators. Molen creates FIFOs between its HW operators (of which no details have been published). CHiMPS first translates the C input into the intermediate representation CTL and then directly exports the required operators as HDL code (operator internals have not been published).

Our own compiler COMRADE has been relying on the GLACE library [25], which provides a very flexible bi-directional interface [17] between compiler and module library: The compiler could not only request the generation of technology-optimized pre-placed netlists, but also receive information on area, throughput/latency/delay as well as control protocols. However, despite the flexibility and performance of GLACE modules, their development was a painstaking manual process that had to be performed for each new target technology. With improvements in automatic synthesis and layout tools, this effort is no longer generally worthwhile (this will be further discussed in Sec. 7). GLACE modules also lacked a standardized control interface, instead describing the existing interface to the compiler and having it generate an appropriate micro-sequencer [16]. In practice, the flexibility offered in this fashion was never truly exploited and needlessly complicated the compiler back-end (which theoretically had to cope with arbitrarily complex control protocols).

None of the HW generation approaches discussed above support as wide a spectrum of both operator functionality as well as advanced features such as operator cancellation (motivated in the next Section). However, it is relatively simple to encapsulate existing generators in Modlib-conforming wrappers. We have already demonstrated this with various netlists created by the Xilinx CoreGen tool.

LMEM is our implementation of memory localization. Similar to [30], it allows the user to explicitly request on-chip storage of arrays. Then, using an approach similar to [22], LMEM integrates all distributed memories into the system memory map to make them accessible to the CPU. We have refined upon the prior work by adding a dedicated hardware unit for paging data between main and localized memories. In Sec. 7, we will examine how the Modlib HW operators interact with the high-bandwidth storage enabled by the local memories in context of a real application.

3. SPECULATIVE PREDICATED EXECUTION

Since support for dynamic scheduling with predicated speculative execution is one of the main advances in Modlib over GLACE, this section gives a brief overview of how a compiler can actually exploit these features. We will use our COMRADE framework as an example.

The RCU HW kernels generated by COMRADE are dynamically scheduled (to reach COMRADE's aim of efficiently compiling even control-intensive irregular code). Fundamentally, this is achieved by a classic data flow model with activate tokens (ATs) indicating the presence of data items on operator inputs. Control flow is modeled by optionally predicating the execution of operators (shown by an incoming control edge in Fig. 1). Speculative execution is supported and occurs when data predecessors already have ATs, but the control predecessor is not yet ready.

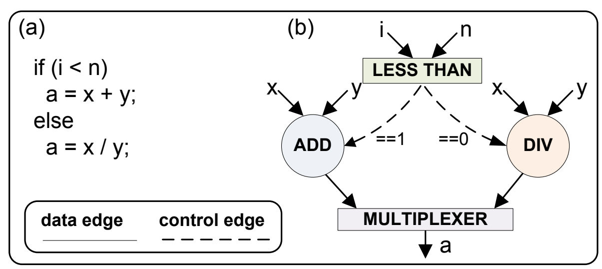

Fig. 1. Speculative execution of alternative branches of an if: (a) C source; (b) speculative predicated data flow.

For the sample code in Fig. 1(a), this is shown in Fig. 1(b). The multiplexer (mux) merges the two alternative datapath branches, with the control condition (the less than operator) allowing only one path's AT to advance to the mux. The mis-speculated result on the other branch has to be eliminated. COMRADE explicitly models this deletion of data using cancel tokens (CTs) which erase ATs (and the associated data item) when they meet. Since COMRADE uses dynamic scheduling, tokens must be buffered to keep execution of the different paths in sync. Modlib supports two such mechanisms, both of which can be used by COMRADE.

In the *static* case, CTs *wait* for associated ATs at the end of each alternative branch (see Fig. 2(a)). Modlib implements this behavior by generating token buffers in the predicated operators if requested by the compiler back-end. Their use allows the datapath to compute ahead of the con-

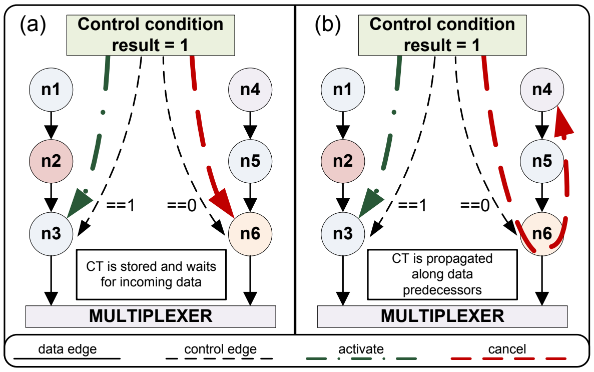

Fig. 2. Token flow for (a) static CTs and (b) dynamic CTs.

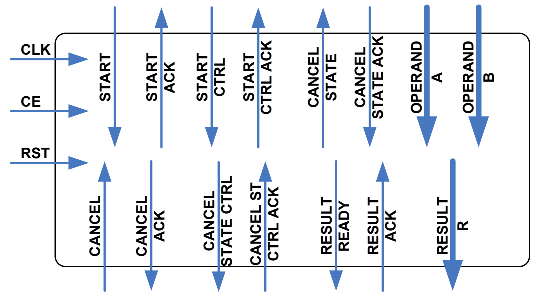

Fig. 3. Modlib Module Interface

trol evaluations, e.g., to speculate across loop iterations.

Alternatively, CTs may be *propagated* in reverse data flow direction to meet incoming ATs. In this *dynamic* case, the token handshaking protocol becomes more complex (e.g., AT and CT enter an operator simultaneously). However, since the CTs are now also moving, they have a greater potential to abort the execution of an already executing operator whose result-to-be would be eliminated anyway. Similarly, an idle operator in the mis-speculated branch would not even be started and the CTs just passed upwards to its data predecessors. We generalize both behaviors with the term *operation cancellation* when scheduling using dynamic or static CTs. A more detailed discussion of the token flow model would exceed the scope of this paper.

4. MODLIB

Implementation-wise, Modlib consists of technology-independent Verilog modules, mostly containing RTL descriptions. All modules provide a flexible universal interface and are highly parameterized and thus suitable for a broad range of HW compilers.

All ANSI-C operators, including 32-bit and 64-bit integer as well as floating point and type conversion operators are supported. For compilers with advanced word-width analysis capabilities, other widths may also be specified. The library is easily extensible, both in terms of adding new implementations for existing operators as well as new operators themselves. This also extends to wrapping externally generated technology-dependent IP blocks in Modlib interfaces. For Xilinx devices, Modlib encapsulates optimized division, modulo, floating-point and type-conversion generated by the CoreGen tool in this manner. Other target technologies could be treated in a similar fashion.

Beyond C-level operators, the library also contains primitives required for datapath synthesis. These include the buffer module nop, the multiplexer module mux and the constant module const. For dynamic scheduling, special modules allow token combination (and, or) and token generation (initial_op, always_op).

Modlib also wraps the interface from the HW kernel to the rest of the system in a portable manner: I/O register modules inreg and outreg are automatically memory-mapped into the CPU address space and allow data exchange under software control. Similarly, the HW datapath can request software intervention by instantiating and writing to an irqreg module. Memory accesses are encapsulated in memread and memwrite modules.

The parameters supported by the Modlib modules can roughly be divided into three groups. The first affects the operator function directly and encompasses the bit-width, or signedness of data or the number of inputs and select scheme (one-hot or encoded) of a multiplexer. The second group deals with the buffering of data, allowing the insertion of transparent queues or fixed-depth shift registers. The third group controls token handling and can indicate static or dynamic CTs and predication. In all cases, if a feature is not required, it will not be generated in HW. Thus, Modlib is also applicable for building simple statically scheduled datapaths.

The main advantage of Modlib over its predecessors, though, is a very flexible customizable control interface, available regardless of the combinational or sequential nature of an operator, its pipelineability, or its actual implementation (RTL or netlist). This allows Modlib to be used in the wide spectrum of HW compilation strategies mentioned in Sec. 1 and will be examined in greater detail next.

4.1. Full Module Interface

In addition to the customary clock, enable and reset signals, the full Modlib interface (shown in Fig. 3) has zero or more data inputs. For unary operators, this will always be the port A, binary operators use A and B, n-ary operators ($n > 2$) pack all inputs into a wider A port by concatenating their bit-widths. Output data (if any) is available at the R port.

When all predecessors an operator is data- or memory-dependent on have completed, asserting the START input starts the operator. In a dynamically scheduled system,

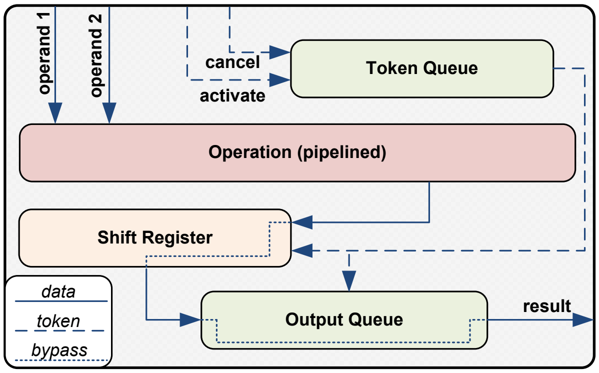

Fig. 4. Modlib Module Implementation

START could, e.g., be connected to an and operator combining all ATs incoming from data and memory predecessors. START_ACK will go high to acknowledge the started state. Note that since only data and memory dependences are considered here, the operator may be started speculatively.

Satisfied control dependences are handled similarly using the START_CTRL and CANCEL inputs. Assertion of START_CTRL indicates a satisfied control condition. The operator is now executing non-speculatively (if it is already running) or will be started non-speculatively once the data and memory dependences are satisfied. START_CTRL_ACK acknowledges the assertion of START_CTRL in the usual manner.

Asserting the CANCEL input indicates an unsatisfied control condition (expressed as a CT) and has multiple possible effects: If it is asserted when an operator is already executing speculatively, it terminates the operator without a result being output. If the operator has already completed and buffered a result, that result is discarded. If the operator has not yet been started, its data predecessors can be informed that *their* results will no longer be required by the canceled operator when it asserts the CANCEL_STATE output. If the canceled operator was the only successor of its predecessors, their operation can also be terminated or be prevented from even starting. The predecessors' acceptance of the CT is indicated by all of them asserting (via an and operator) the CANCEL_STATE_ACK input. Only then is the CT removed from the current operator, otherwise it stays buffered. This will also be the case if the compiler back-end requests a HW kernel without the backward propagation of CTs (dynamic CTs), instead keeping them stationary in the canceled operator and waiting for incoming ATs to extinguish (static CTs). As shown in Sec. 7, dynamic CTs can lead to shorter execution times, but static CTs requires less HW area. An asserted CANCEL_ACK output indicates that the operator has accepted the incoming CT.

The operator will assert the RESULT_READY output to indicate the availability of a result on the R output, in effect creating an AT. The corresponding input RESULT_ACK indicates that all data successors have consumed the result (an and operator combining their START_ACK outputs) and it no longer needs to be buffered in the operator.

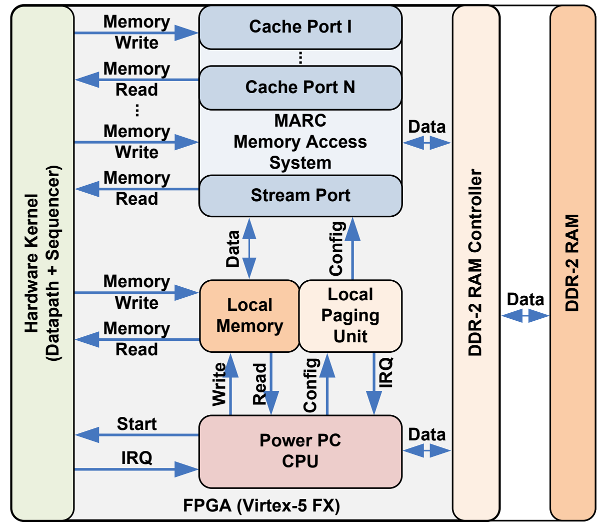

Fig. 5. Experimental Target Platform Virtex-5

In nested control structures, canceled condition operators propagate the CT to nested conditions in the hierarchy by asserting the CANCEL_STATE_CTRL output, acknowledged as usual using CANCEL_STATE_CTRL_ACK. This allows the cancellation of many levels of speculative execution at once when a high-level condition has been evaluated [8].

Modules interfacing with the rest of the system (CPU, shared memories) of course have other, platform-specific ports connecting it to central processor buses, interrupt, or memory controllers.

4.2. Module-internal Structure

Fig. 4 gives an overview over some of the internal structures of a Modlib module wrapper. The intrinsic operation (e.g., an RTL operator or embedded IP block) consumes the incoming operands. Its output can optionally be buffered in a transparent output queue to decouple the execution of the module from stalled data successors. Also optionally, the output can be delayed for a fixed number of cycles in a configurable shift register. This can be used both for balancing pipeline paths with unequal latency as well as enabling higher clock-frequency operation by retiming in the logic synthesis tool. Again, if these features are not requested by the user (the compiler back-end), they will be optimized away in synthesis.

5. TARGET ARCHITECTURE

Modlib has so far been used on target platforms with Xilinx Virtex II Pro and 5FX FPGAs. The evaluation in Sec. 7 is based on experiments on the 5FX-based Xilinx ML507 development board. For purposes of this discussion, the architecture (shown in Fig. 5) consists mainly of the HW kernel, the CPU (PPC440), and shared memories (both BlockRAM on- and DDR2-SDRAM off-chip).

All accesses to the external memory are performed through the flexible MARC interface [19], attached by the high-bandwidth low-latency FastLane+ back-end [20] to the SDRAM. MARC provides the HW kernel with multiple memory ports, offering both caches for efficient irregular, and buffered streams for efficient regular access patterns. The HW kernel itself consists of the instantiated Modlib modules for the operators as well as a controller (we examine different control schemes in Sec. 7).

The platform runs under a full-scale Linux operating system. Since the HW kernel has full master-mode access to the SDRAM main memory, it just needs to receive a few words of parameter data from the CPU under software control and then handles memory data on its own. It indicates the completion of the operation (or requests a service such as memory allocation from the CPU) using the FastPath low-latency signaling scheme [20].

To improve memory bandwidth and latency even further, the architecture also integrates FPGA on-chip memory shared between CPU and HW kernel. The CPU can explicitly request the high-speed paging of data between these on-chip memories and the main memory, also using a dedicated HW block (Local Paging Unit, LPU). This infrastructure will be used to make localized memories available to the compile flow and is discussed in the next Section.

6. MEMORY LOCALIZATION

Prior to this work, COMRADE-generated HW kernels relied solely on MARC/FastLane+ for high-performance access to system memory (shared with the processor). Up to four simultaneous memory accesses can be handled by this setup, which completely exploits the physical bandwidth of the used memory chips. However, aggressive loop parallelization and pipelining techniques often require even more bandwidth to achieve the optimum initiation intervals.

Memory localization works around the bottleneck of a single shared memory by providing the HW kernel with parallel accesses to independent memories (see Sec. 5 for the actual HW architecture). These on-chip memories, while small compared to the main memory, can be accessed by the HW kernel with a much larger degree of parallelism. The processor can also access these on-chip memories, both using software-programmed reads/writes, as well as initiating bulk data transfers between them and from/to main memory using the Local Paging Unit (LPU).

COMRADE is exploiting these capabilities using the LMEM subsystem. As shown in Sec. 7, the speed-ups for a non-trivial sample application are significant. The LMEM prototype implemented for this work currently has three limitations: First, the programmer has to indicate in the source program which data structures (specifically, arrays) are to be localized in the on-chip memories. Second, at most one read and one write access in the code is allowed per localized array. Third, the LPU is explicitly programmed to initialize the localized memories from main memory before starting the HW kernel, and to write back modified data after it has completed. Future work will thus be directed at making LMEM more intelligent and automatically perform the necessary analyses and code generation.

7. EXPERIMENTAL RESULTS

A. Performance Overview

We will first compare Modlib with both static and dynamic CTs to GLACE using six benchmarks compiled from C. GfMultiply is part the Pegwit elliptic curve cryptography application. susan performs edge detection on gray scale images, sha_3 realizes the Secure Hash Algorithm. They are part of the MiBench [24] and CHStone [10] suites, respectively. fcdf22_2 computes a 2-D wavelet transform for image compression [18]. To test specific features, we use the synthetic kernels memcpy_8 (array copy, unrolled x8) and vecmult_10 (vector multiplication, unrolled x10).

For each the examples, Tab. 1 shows the simulated execution time, area requirements (shown separately for the required control logic) as well as post-P&R maximum clock speeds. The synthesis and mapping tools used were Synopsys Synplify Premier DP 9.6.2 and Xilinx ISE 11.4. All kernels easily achieve the default system clock frequency of 100 MHz of our ML507-based platform. Furthermore, note that despite its limited portability and flexibility, versions of the kernel using hand-optimized GLACE modules could achieve maximum clock speeds on average 8.8% faster than those of the fastest Modlib configuration tested (using static CTs). However, since the lower clock speed of the rest of the system components (CPU, PCI and memory controller, etc.) sets an upper bound on the kernel clock speed, the speed advantage of GLACE alone is not relevant when considering the system-level performance. Area-wise, Modlib modules (with static CTs) are in average 6.7% smaller than their GLACE counterparts since inter-module synthesis can take greater advantage of boundary optimization than possible on the embedded structural netlists.

For comparing the different sequencing schemes, we now consider the control overhead of the kernels separately (disregarding the datapath). Going to dynamic CTs, control

Kernel	Runtime			FPGA Area						Max. Freq. (MHz)		
	#Cycles			Total #Slices			Sequencer #Slices					
	dCT	sCT	GL	dCT	sCT	GL	dCT	sCT	GL	dCT	sCT	GL
gfMultiply	1688	1688	1656	972	775	1106	162	98	241	157	174	131
susan	453	464	454	3462	2442	3496	651	350	498	107	108	110
sha_3	501	501	537	1208	897	911	180	72	132	131	157	183
fcdf22_2	1591	1591	1591	1268	975	968	198	105	134	132	146	178
memcpy_8	380	380	378	1621	1276	1104	214	123	141	149	147	200
vecmult_10	247	247	245	2249	2110	2009	340	203	319	118	119	120

Table 1. Kernel runtime, area consumption and maximum frequencies for Modlib with dynamic CTs (dCT), static CTs (sCT), and GLACE (GL) on a Virtex-5 FX.

area increases on average by 23% over the GLACE version. However, the dynamic CT-sequencer in Modlib is significantly more powerful and decouples the operators of converging data flows (see next Section). Using static CTs in Modlib, the controller area shrinks by 34% compared to GLACE, while still allowing the looser coupling. When directly comparing static and dynamic CT-controllers in Modlib, the first are on average 21% smaller and 7% faster (in terms of max. clock speed).

For these examples, the different techniques take very similar numbers of clock cycles to execute. We will consider the impact of the different techniques for variable initiation intervals (due to different length control paths, Sec. 7.B) and loosely coupled datapaths (Sec. 7.C) next.

B. Dynamic CTs vs. Static CTs

Dynamic CTs can speed-up execution over static CTs when a computation has converging paths of differing lengths (e.g., very short if, very long else). Such an occurrence can be induced e.g., by nested ifs, irregular control break/continue, or by a high-latency operator in one of the parallel paths. While this could be compensated for by balancing all paths, it quickly becomes inefficient if there are many such irregularities (common in general-purpose C code): All loop initiation intervals would be set to the length of the longest path. Fig. 6 gives an example, using a slow divider with 34 cycles of latency in the else path.

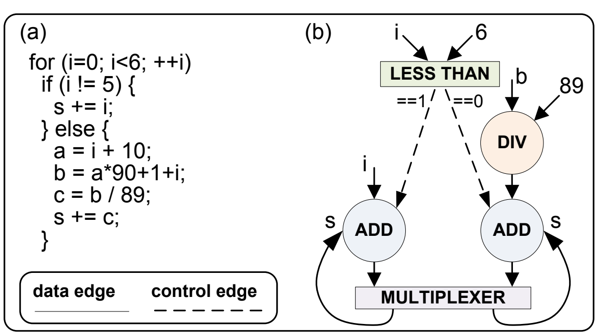

Fig. 6. Mismatched length in converging paths: (a) C source; (b) hardware.

When using dynamic CTs, the divider is executed only for $i = 5$, and canceled in all prior iterations, leading to a run-time of 70 cycles for the entire loop as a hardware kernel. Static CTs, while allowing smaller and faster modules, also cancel mis-speculated computations less aggressively (they remain stationary at mux inputs). The mux becomes available for new data only if prior results incoming on *all* of its input branches have either been selected or canceled. In this case, the total runtime would increase to 101 cycles, a slow-down by 44% over dynamic CTs. Since Modlib supports both styles of cancelling, dynamic CTs can be used just where advantageous, with static CTs being the default for low-complexity code.

C. Transparent Output Queues

Since Modlib contains variable-latency operators (e.g., cached memory accesses), traditional static pipeline balancing methods cannot always be used. Instead, Modlib operators can be parametrized to insert transparent queues into their output to allow for a looser coupling between operators. These queues can be very efficiently synthesized onto the primitive shift register blocks on most modern FPGAs. As a simple example for their impact: In an image processing application (contrast-stretching), even enabling very shallow queues of just 16 entries alleviates the effect of cache misses by allowing only part of the datapath to stall, leading to a 17% speed-up (116 vs. 136 cycles per loop iteration).

D. Combining Modlib and LMEM

To study the various techniques in context of a larger hardware kernel, we examine the MD5 message digest algorithm [29]. In brief, MD5 handles a message in 512b chunks. Each chunk is then processed in four rounds, each having 16 computation steps. Due to the block-chained nature of the algorithm, blocks from a single message can only be processed sequentially. However, if multiple independent messages have to be processed (e.g., for a cryptanalytic attack), parallelism, e.g., in the form of pipelining could be used to improve throughput, which can be estimated using Eq. 7.1.

$$\text{Throughput [Mbps]} := \frac{512\text{b} \cdot \text{Frequency [MHz]}}{\text{\#Cyc. per Chunk}} \quad (7.1)$$

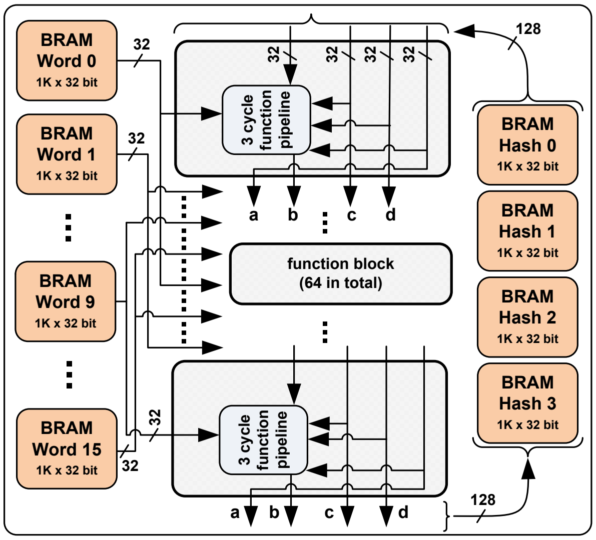

Fig. 7. MD5 Modlib HW Kernel Architecture

We use COMRADE to compile the processing of the 512b chunks into an accelerated hardware kernel, using loop unrolling to create 64 function blocks, each containing the appropriate three-cycle sub-pipeline for computing one of the intermediate hash values a through d for one of the 16 32b words making up the entire message. Fig. 7 illustrates the generated architecture.

When using pipelining to process different messages in parallel, the datapath requires access to 16 32b words of data each clock cycle for optimum throughput. Since the main memory on the ML507 platform cannot deliver at this rate, we advise COMRADE to localize the messages in fast on-chip BlockRAM using the LMEM infrastructure. Specifically, we request on-chip storage for up to 1024 messages. Since the corresponding hashes are also localized in BlockRAMs, we require a total of 20 BlockRAMs. BlockRAMs are dual-ported, we can thus run the Local Paging Unit in parallel to the hardware kernel to keep the BlockRAMs filled with new messages. We have also enabled transparent output queues in the modules to improve throughput.

The speed-up of using queues over an implementation without them is $S_k = \frac{n \cdot k}{3 \cdot k + n - 3}$, where k is the number of messages to process, $n = 64 \cdot 3 = 192$ is the pipeline latency and 3 is the minimum initiation interval of the pipeline. The selected queue depth q is an upper bound for k, until $q \geq n$ (the buffers are deep enough to hold the entire pipeline length). Then, an arbitrary number of messages can be processed pipeline-parallel without a drop in throughput. Of course, throughput would also drop if the LPU takes longer to refill the localized memories with k new messages than the kernel requires to process a set of k messages.

Tab. 2 shows the achievable throughput for different k. Note that we have scaled q to keep up with k. While the increasing complexity of the queues leads to a drop in clock speed, the throughput steadily improves. Furthermore, we need an additional 14 cycles of pipeline latency for HW/SW communication, increasing the total latency to 206 cycles. Also, for this kernel architecture, increasing k is not the only way to run the pipeline at full throughput. Once k exceeds 64 (the number of function blocks), successive chunks of the *same* message could be fed to the pipeline at full speed (since the loop carried data dependence on the prior chunk's intermediate hash values is now satisfied). This is also known as C-slow execution [21], here with $C = k = 64$.

When decreasing q below the pipeline length, execution times will increase dramatically, with a corresponding drop in throughput. As seen in Fig. 8, the slow-down for $k = 1024$ would be 1.97 for $q = 32$, and 62.28 for $q = 0$ (not using any queues).

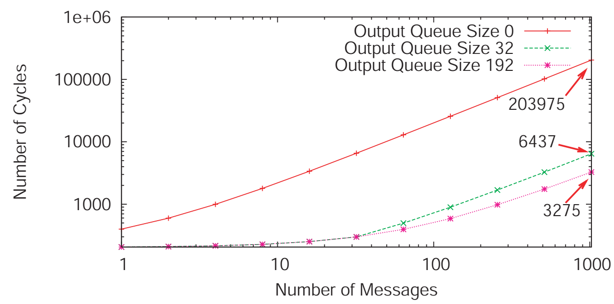

Fig. 8. MD5: runtimes for different output queue sizes.

Tab. 2 compares the performance of a number of COMRADE-generated hardware kernels for Modlib with dynamic and static CTs, for strictly sequential or pipelined operation, and with different queue depths. Unfortunately, only the kernels with a queue depth of 0 fit into our target FPGA (the Virtex-5 XC5VFX70T), which offers up to 11,200 slices. For greater queue depths up to 128, the largest available Virtex-5 VFX FPGA (the XC5VFX200T) with 30,720 slices would suffice. Although the largest kernel (having a queue depth of 192) does not require more slices in total than offered by the XC5VFX200T, the mapping report here shows an overmapping of distributed memory LUTs (36,480 available, 44,423 required). However, shortage on FPGA area will not be problem in the near future: the existing Virtex-6 and the announced Virtex-7 FPGAs offer up to 120k and 300k slices, respectively. To be able to compare our results on a Virtex-5 basis, we show estimated Virtex-5 values for queue depth 192. The number of slices presented is estimated from the LUT and FF requirements as shown in the interim mapping report; for this estimation we assume the same packing density obtained for queue depth 128. We further assume the frequency to match the queue depth 128 value of 100 MHz.

Interpreting the results shown in Tab. 2, the very regular MD5 algorithm does not profit from dynamic CTs (here,

Design	Device	#Messages	Runtime [Cycles]	Speed-up S_k	Queue Size	Freq. [MHz]	#Slices [V5 eqv.]	Latency	Throughput [Mbps]	Throughput [Mbps/Slice]
dyn. CT sequential	Virtex-5	1	205	1.0	0	120	10,204	192	300	0.03
dyn. CT pipelined	Virtex-5	16	247	13.0	16	100	19,689	192	3,317	0.17
static CT sequential	Virtex-5	1	205	1.0	0	134	8,584	192	335	0.04
static CT pipelined	Virtex-5	16	251	13.0	16	120	15,491	192	3,916	0.25
static CT pipelined	Virtex-5	32	299	21.6	32	119	15,946	192	6,521	0.41
static CT pipelined	Virtex-5	64	395	32.3	64	120	16,752	192	9,955	0.59
static CT pipelined	Virtex-5	128	587	42.9	128	100	20,266	192	11,165	0.55
static CT pipelined	Virtex-5	1,024	3,275	60.3	192	100*	26,976*	192	16,009*	0.59*
MD5_32_u_pipe [32]	Stratix II	32	65	16.7	-	66	11,957	34	32,035	2.68
Heliontech IP [11]	Virtex-5	1	66	-	-	174	279	66	1,349	4.84

Table 2. MD5 HW Kernel performance on Virtex-5 and performance comparison. *Values for queue size 192 are estimated.

they just increase area and slow down the clock speed). But dynamic scheduling with static CTs is competitive with carefully hand-optimized designs for MD5 [32] [13] [4] [14] [11]. A MD5 SW implementation on a 3 GHz Pentium 4 processor achieved 165 Mbps and was outperformed by all investigated HW implementations at much lower frequency. For brevity, we just show the designs with maximum throughput (32,035 Mbps) and most efficient area use (4.84 Mbps/slice). Altera Stratix II ALMs were converted to Virtex-5 slices using the formulae in [1].

Our own best automatically compiled kernel achieves 16,009 Mbps with an area efficiency of 0.59 Mbps/slice. In these terms, we achieve 12 % of the performance of the hand-optimized Heliontech design. Regarding the throughput, our best kernel achieves 50 % of the performance of the hand-optimized MD5_32_u_pipe design.

8. CONCLUSION AND FUTURE WORK

We examined some of the low-level infrastructure for high-level language to HW/SW co-compilers in context of the COMRADE framework. Modlib, which is publically available at [9], has proven quite successful in being much easier to maintain while offering far greater flexibility than our previous approach of GLACE. It can achieve a similar quality of results (when restricted to a similar feature set), but is much more adaptable to both to different scheduling and execution models, as well as different target technologies. Modlib is currently used not only in C to HW compilation, but also in a compiler generating deep computing pipelines for Geometrical Algebra computations [12].

The new LMEM infrastructure for memory localization has also been successful in obtaining first experimental results on possible speedups on our target platform. The compiled MD5 accelerator has performance comparable to highly optimized manual implementations. Future work will focus on easing the use of LMEM by automating both the analysis as well as synchronization code generation steps (programming the LPU).

References

[1] 1-Core Technologies. FPGA Logic Cells Comparison. `http://www.1-core.com/library/digital/fpga-logic-cells/fpga-logic-cells.pdf`, 2010.

[2] M. Budiu. *Spatial Computation*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, USA, 2003.

[3] J. Cong, Y. Fan et al. Platform-Based Behavior-Level and System-Level Synthesis. In *Proc. Int. SOC Conf.*, pp. 199–202. 2006.

[4] J. Deepakumara, H. M. Heys et al. Fpga Implementation Of Md5 Hash Algorithm, 2001.

[5] H. Gädke and A. Koch. Accelerating Speculative Execution in High-Level Synthesis with Cancel Tokens. In *ARC*, vol. 4943/2008 of *LNCS*, pp. 185–195. Springer, 2008.

[6] Z. Guo, W. Najjar et al. Efficient Hardware Code Generation for FPGAs. *ACM Trans. Archit. Code Optim. (TACO)*, 5(1):1–26, 2008.

[7] S. Gupta, N. Dutt et al. SPARK: A High-Level Synthesis Framework for Applying Parallelizing Compiler Transformations. In *VLSI*, pp. 461–466. 2003.

[8] H. Gädke, F. Stock et al. Memory Access Parallelisation in High-Level Language Compilation for Reconfigurable Adaptive Computers. In *FPL*, pp. 403–408. 2008.

[9] H. Gädke-Lütjens, B. Thielmann et al. Modlib Download Page. http://www.esa.cs.tu-darmstadt.de, Section Research/Publications.

[10] Y. Hara, H. Tomiyama et al. Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis. *Journal of Information Processing*, 17:242–254, 2009.

[11] Helion Technology Limited. Helion Technology. Datasheet, High Performance MD5 Hash Core for Xilinx FPGA, 2008.

[12] D. Hildenbrand, J. Pitt et al. *Geometric Algebra Computing for Engineering and Computer Science*, chapter High Performance Geometric Algebra Computing based on Gaalop. Springer, to appear 2010.

[13] A. T. Hoang, K. Yamazaki et al. Multi-stage Pipelining MD5 Implementations on FPGA with Data Forwarding. In K. L. Pocek and D. A. Buell, editors, *FCCM*, pp. 271–272. IEEE Computer Society, 2008.

[14] K. Jarvinen, M. Tommiska et al. Hardware Implementation Analysis of the MD5 Hash Algorithm. In *HICSS*, p. 298.1. IEEE Computer Society, Washington, DC, USA, 2005.

[15] N. Kasprzyk and A. Koch. High-Level-Language Compilation for Reconfigurable Computers. In *ReCoSoC*. 2005.

[16] A. Koch. Compilation for Adaptive Computing Systems Using Complex Parameterized Hardware Objects. *Journal of Supercomputing*, 21:179–190, 2002.

[17] A. Koch. FLAME Library Specification. Technical report, Tech. Univ. Braunschweig (Germany), Dept. for Integrated Circuit Design (E.I.S.), 2004.

[18] S. Kumar, L. Pires et al. A benchmark suite for evaluating configurable computing systems—status, reflections, and future directions. In *FPGA*, pp. 126–134. ACM, New York, NY, USA, 2000.

[19] H. Lange and A. Koch. Memory Access Schemes for Configurable

Processors. In *FPL*, pp. 615–625. 2000.

[20] H. Lange and A. Koch. Architectures and Execution Models for Hardware/Software Compilation and their System-Level Realization. *IEEE Trans. on Computers*, 99(PrePrints), 2009.

[21] C. E. Leiserson, F. M. Rose et al. Optimizing synchronous circuitry by retiming. In *Proc. Caltech Conf. on VLSI*, pp. 87–116. 1983.

[22] M. Luthra, S. Gupta et al. Interface Synthesis using Memory Mapping for an FPGA Platform. In *ICCD*, vol. 0, p. 140. IEEE Computer Society, Los Alamitos, CA, USA, 2003.

[23] M. Bowen, Embedded Solutions Ltd. Handel-C Language Reference Manual. `http://www.pa.msu.edu/hep/d0/l2/Handel-C/Handel%20C.PDF`, 2009.

[24] M. Guthaus et al. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proc. IEEE Int. Workshop on Workload Characterization*, pp. 3–14. 2001.

[25] T. Neumann and A. Koch. A Generic Library for Adaptive Computing Environments. In *FPL*, vol. 2147/2001, pp. 503–512. 2001.

[26] E. M. Panainte. *The Molen Compiler for Reconfigurable Architectures*. Ph.D. thesis, Technical University Delft, 2007.

[27] Y. Panchul, D. Soderman et al. System for Converting Hardware Designs in High-Level Programming Languages to Hardware Implementations. `http://www.freepatentsonline.com/20010034876.pdf`, 2001.

[28] A. Putnam, D. Bennett et al. CHiMPS: A C-level compilation flow for hybrid CPU-FPGA architectures. In *FPL*, pp. 173–178. 2008.

[29] R. L. Rivest. The MD5 Message Digest Algorithm. RFC 1321, 1992. `ftp://ftp.rfc-editor.org/in-notes/rfc1321.txt`.

[30] Robert Schreiber et al. PICO-NPA: High-Level Synthesis of Nonprogrammable Hardware Accelerators. *Journal of VLSI Signal Processing*, 31(2):127–142, 2002.

[31] Synfora. Synfora Website. `http://www.synfora.com`, 2010.

[32] Y. Wang, Q. Zhao et al. Ultra High Throughput Implementations for MD5 Hash Algorithm on FPGA. In *HPCA*, pp. 433–441. 2009.