High-Level Synthesis of Resource-Shared Microarchitectures from Irregular Complex C-Code

Björn Liebig, Andreas Koch
Embedded Systems and Applications Group (ESA), Technische Universität Darmstadt
Email: {bl,ahk}@esa.cs.tu-darmstadt.de

Abstract—
Many high-level synthesis (HLS) tools aim at the hardware translation of input programs with relatively short loop bodies, often with a very regular control flow. However, codes from domains such as control engineering and numerical simulation often have a considerably different structure with large loop bodies holding (tens of) thousands of individual operations. Compilation of such codes not only requires the sharing of hardware operators, but also the efficient storage and forwarding of many intermediate results. Both academic as well as industrial synthesis tools have great difficulty coping with such input programs.

We present Nymble-RS, a C-to-hardware compiler optimized to translate such complex programs. When evaluated on codes for the domain of convex solvers, the generated accelerators reach clock frequencies of over 200 MHz (exceeding those achieved by a state-of-the-art industrial tool by more than 3x), and offer speed-ups of up to 5x over software executing on the 800 MHz Cortex-A9 CPUs used in typical reconfigurable system-on-chips.

I. Introduction

Many HLS tools aim at the translation of relatively short input programs to hardware. Often, these algorithms (e.g., from signal and image processing applications) also have a very regular control flow, consisting of one more small-bodied loop nests that are often amenable for optimizations such as unrolling and pipelining.

However, codes from domains such as control engineering and numerical simulation often have a considerably different structure with large loop bodies holding (tens of) thousands of individual operations (often floating point). Compilation of such codes not only requires the sharing of hardware operators, but also the efficient storage and forwarding of many intermediate results. Both academic as well as industrial synthesis tools have great difficulty coping with such input programs.

We have thus concentrated our efforts on creating a HLS system capable of translating this type of irregular "C"-code into accelerators executing on FPGAs. Using LLVM as the front- and mid-end (providing target-independent optimizations), and building on our HLS back-end Nymble [1], we have created Nymble-RS, a HLS engine aggressively exploiting resource sharing to fit the required computations on FPGAs. As a target architecture, we aim for a platform containing one or more software-programmable processors tightly coupled to reconfigurable logic. Such chips are readily available from multiple manufacturers (e.g., Xilinx Zynq, Altera SoCs etc.).

978-1-5090-5602-6/16/$31.00 ©2016 European Union

We evaluate our approach by compiling solver codes for complex optimization problems into hardware. While these "C" programs are light in control flow, they contain many thousands of double-precision floating point operations. This structure is unsuitable for a direct compilation to the purely spatially distributed model of computation (one hardware operator per operation) commonly used on FPGAs, as the resulting hardware would have excessive chip area requirements. Also, the solver codes employ pointer-based data structures that some HLS systems also cannot deal with.

We compare our approach with other academic and industrial "C"-based HLS systems as well as with the software performance achieved on high-performance embedded hard-core processors (e.g., 800 MHz ARM Cortex-A9 including a hardwired double-precision floating-point unit (FPU)).

The key contributions presented in this paper are 1) the use of distributed and partitioned microcode to control multiplexers for resource sharing, 2) a spilling mechanism to reduce register usage in hardware data paths, and 3) a case study using the developed tool to accelerate solvers for convex optimizations targeting the Xilinx Zynq platform.

After looking at related work (Section II), we give a brief overview of convex optimization to show the nature of the computations managed by our tool (Section III). Section IV presents the required techniques we had to employ to create area-optimized hardware implementations for the complex, highly irregular solver codes. Among the technologies discussed here are the synthesis of controllers using distributed hierarchical micro-code, data transport optimizations trading-off between shift registers, local memories, allocation of dedicated registers, as well as an extended schedule-dependent tree-height reduction. The impact of the different techniques is evaluated in Section V and summarized in Section VI with a brief look towards future work.

II. Related Work

A. High-Level Synthesis

An increasing number of HLS tools, originating both from industry and academia, is capable of translating (often differing) subsets of C into synthesizable RTL HDL code.

Commercial tools include Mentor Catapult[2], Xilinx Vivado HLS [3] and Synopsis' Synphony C Compiler [4]. All of these tools have in common that they provide only hardware synthesis, they do not consider hardware/software-co-execution and interfacing in a heterogeneous system such as an ACS. Hardware/software co-synthesis is supported in academic tools, e.g., in Nymble [1], LegUp [5], ROCCC [6], COMRADE [7], and Garp CC [8].

Garp CC [8] offers automatic partitioning to execute a C program on a MIPS processor augmented with a coarse-grained reconfigurable array (CGRA). It is based on the SUIF compiler infrastructure [9] and uses a modified GCC tool chain. In addition to high level synthesis, it also provides hardware/software partitioning without user intervention.

COMRADE, also based on SUIF, focuses on the compilation of control-intensive C code into dynamically scheduled accelerators [7]. The proposed compute model includes very finely granular hardware/software partitioning, with the proposed architecture allowing master-mode accesses to memory shared between the reconfigurable compute unit (RCU) and the software-programmable processor (SPP).

Nymble [1] extends many concepts of COMRADE while continuing to offer finely granular hardware/software partitioning. It uses the LLVM compiler framework as front-end and for target-independent optimization. The generated hardware kernels and software parts execute together in a shared memory architecture in the same address space (allowing transparent passing of pointers between software and hardware). The MARC II configurable memory system [10] is used to perform multiple concurrent read/write accesses. In contrast to COMRADE, Nymble creates pipelined statically scheduled accelerators, moving the focus away from control-intensive applications to data-driven computation.

The C to hardware compilers ROCCC and LegUp are also based on the LLVM compiler framework. While ROCCC lacks support for many commonly used C constructs, e.g., pointers and variable-distance shifts, LegUp supports a large subset of C-code. ROCCC can be seen as a specialized tool, providing good results on a smaller subset, while LegUp does well even in the general case.

A limitation that LegUp shares with many other compilers is the partitioning granularity: Only complete functions are translated into hardware. Nymble is more selective: It can translate just part of function to hardware, and even exclude sections within that area, moving them back to software, as required. For LegUp, the partitioned local address spaces between CPU and FPGA also complicate the handling of pointer-based data structures and often require copying and relocation.

B. Resource Sharing

Ideally, reconfigurable computing is performed in a fully spatial model, associating an individual hardware operator with each operation in the algorithm (input program). This approach allows hardware-accelerators to often exceed the performance of software-programmable processors, which reuse (time-multiplex) the same limited number of compute elements for all operations.

However, the fully spatial approach quickly becomes infeasible on current-generation reconfigurable devices when floating point operations are considered. On most FPGAs, appropriate hardware operators have to be composed from many low-level primitives, requiring significantly more chip area than integer operators. In practice, large numbers of such operators cannot be implemented with high throughput and low latency.

Thus, even for reconfigurable computing, resource sharing must be considered for costly floating point operators. However, a number of aspects closely tied to the underlying FPGA architecture have to be considered: First, the fully spatial approach allows the use of operators specialized for each operation (e.g., in terms of bit widths, number formats, or constant inputs). When attempting to reuse operators, this specialization can often only be performed in a more limited fashion and needs special care. One technique extracts recurring operation patterns from the original DFG to execute on the same operator, and accepts small variations in bit widths for a greater degree of reuse [11], [12]. However, the shared operator will then have the maximum of the bit widths required by the operations, and may end up being slower than specialized operators.

Second, reusing operators requires wide multiplexers (64b for double-precision data) on their inputs to connect them to the different data sources. Such multiplexers require significant area and delay, depending not only on the FPGA architecture but also the interface of the shared operator itself [13].

Third, the multiplexers must be controlled over time, establishing the connections required by the execution schedule. In the fully spatial paradigm, a commonly used controller architecture uses a Petri net-like N-hot approach, where the activation state of each operator is controlled by an associated flip-flop, and multiple of these flip-flops may be active in parallel (easily supporting pipelined execution).

However, when attempting to do aggressive resource sharing, this established approach can be inferior to the use of microcoded controllers. Microcode has often been used to drive hardwired compute elements, but mainly in HLS for ASICs and ASIPs [14]. In Section IV-B, we present a refined scheme better suited for mapping to FPGAs.

III. Convex Optimization as Benchmark Domain

This section gives a brief overview of convex optimization, which is a key component of many advanced control systems (e.g., trajectory planning for collision avoidance in autonomous cars) and has been a major motivation for this work. Convex optimization is a subclass of mathematical optimization, which in general. may be defined as follows: Given an objective function $f : \mathbb{R}^n \to \mathbb{R}$, m constraint functions $g_i : \mathbb{R}^n \to \mathbb{R}, i = 1, ..., m$ and m constraints $b_1, ..., b_m \in \mathbb{R}$:

$$\textbf{minimize } f(x)$$
$$\textbf{subject to } g_i(x) \leq b_i, i = 1, ..., m$$

The vector x is also called the optimization variable in this case. A function $h : \mathbb{R}^n \to \mathbb{R}$ is called a *convex* function if it satisfies the inequality $f(\alpha x + \beta y) \leq \alpha f(x) + \beta f(y)$ for all $x, y \in \mathbb{R}^n$ and all $\alpha, \beta \in \mathbb{R}_0^+$ with $\alpha + \beta = 1$ [15].

Many optimization problems can be transformed to a convex optimization problem. E.g., maximizing a concave function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ is equivalent to the convex minimization of $-f$.

The convexity property makes such problems easier to solve than the general case [15], and allows the creation of efficient automatic solvers [16], [17]. CVXGEN, which targets problems that can be modeled as convex quadratic programs, is one such tool which has shown superior performance over competing approaches [18].

The generated C code for the solver consists of many floating point operations, is almost branch-free, and does not make any library calls. It is thus suitable for stand-alone execution in embedded systems lacking complete library support. The solver code relies on three key data structures: The values of the input parameters, the optimization variables, and a work area for temporary intermediate values. They are realized as C structures that contain several arrays of double precision values. In almost all cases, data is addressed directly, with only very few pointers being used. These properties also make the code attractive for compilation to reconfigurable hardware, and its use as highly scalable compute-intensive benchmarks for the evaluation of HLS tools.

IV. Area-Efficient HLS for complex irregular C-Code

Branch-free code is very interesting for hardware synthesis, as the degree of parallelism is now only limited by data dependencies and available resources (no control dependencies exist). However, the relatively large chip area required for individual floating point operators, combined with the large number of operations in typical auto-generated solvers, require *sharing* of operators among multiple operations. This section presents some of the techniques Nymble-RS uses to generate resource-shared hardware implementations of the solvers. As will be seen, fitting the long irregular (not dominated by small-bodied loops) solver computations at all into mid-size FPGAs (40-120K LUTs) requires significant efforts for the tools. Nymble-RS uses static scheduling, but is able to handle variable-latency operations such as cached memory accesses by stalling/restarting the entire datapath. We refer to the statically determined execution order as *stages*, which may differ from the actual clock cycles due to cache misses and other main memory latencies.

Sharing a hardware *operator* between N different *operations* in the Control Data Flow Graph (CDFG) usually results in the creation of a N-to-1 multiplexer for each input. Two aspects have to be considered for this approach: First, especially for integer arithmetic operators, the size of the multiplexer would often exceed the size of the actual operator. Most hardware compilers thus multiplex only operators when we actually save area (e.g., floating point operations, cache ports to main memory etc.). Second, increasing degrees of sharing lead to denser interconnections between data sources (intermediate value storage, outputs of other operators). However, once a connection between a data source and an input multiplexer has been created, the connection can be *reused* for

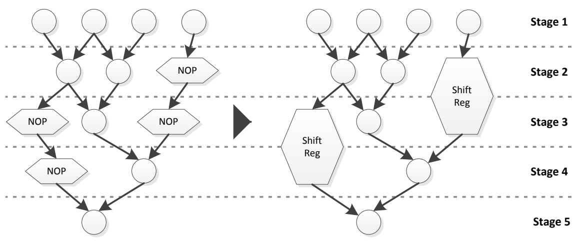

Fig. 1. Reducing the number of NOPs using shift registers

all data originating from that source. Thus, even though an operator is reused for different operations in different stages, the multiplexer width no longer grows. At most, all inputs of all operators are connected to all data sources, leading to a worst case of $O(n^2)$ connections for n operators. While the hardware of each multiplexer is relatively simple even with 64b-wide inputs, the quadratic growth of the interconnection density can lead to place-and-route problems later. Note that intermediate registers (see below), which also act as operators in this model, also contribute to the growth of interconnection density and multiplexer size.

A. Storing Intermediate Values

An intermediate result computed earlier in the schedule has to be retained until the last time (indicated by the number of the schedule step, here called *stage*) it is used, thus defining its *lifetime*. The way these intermediate results are stored significantly affects the required chip area. In a spatial approach, each intermediate result would have a dedicated register connected to its source operator for buffering the value. But this is not practical for larger programs containing thousands of floating point operations, as it would require thousands of 64 bit registers, which would then cause a massive growth in the size of operator input multiplexers.

1) Individual Delay Registers: As a simple solution, intermediate values can be stored-and-forwarded from stage to stage in *individual registers*, which could be modeled as No-Operations (NOPs) in the CDFG. Note that we consider these NOPs to be *operations*, which can then *share* actual hardware registers (which act as *operators* in this case). We use a fast linear-scan register allocator [19] to map NOPs to registers. While feasible, using just individual delay registers would still require too much chip area: Early experiments determined that the datapaths created for the test cases used in this paper have long value lifetimes, and would result in more than 75% of all operations being NOPs.

2) Multi-Stage Shift Registers: Instead of individual registers, we can more efficiently hold values for longer time intervals using *shift registers* as hardware operators. These are expressed in the CDFG as multi-stage NOP operations, and can be mapped very efficiently to FPGA primitives such as SRL16 etc. A shift register of depth k can delay an intermediate value for k stages. We can reuse shift registers to delay values from multiple data sources: as long as their inputs and outputs are accessed in *different* stages, the actual lifetimes may overlap. Fig. 1 multiplexes a single k-stage shift register

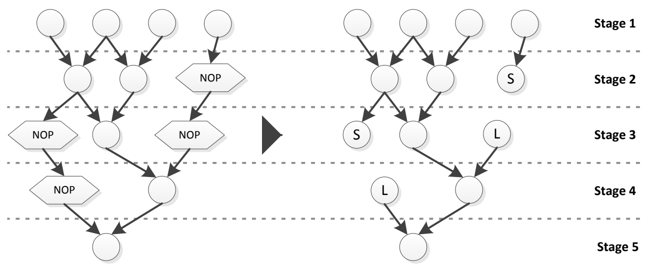

Fig. 2. Two NOP-chains are replaced by store (S) and load (L) operations

to store two values with overlapping lifetimes (since they are enqueued/dequeued at different schedule stages). When using this approach, we choose powers of two as the depths of the predefined shift registers in our operator set, thus allowing the easy composition of longer delays with only few shift register operators.

3) Scratch Pad Memories: If *many* values have to be retained for long lifetimes, even the use of shift-registers requires too much area (see Table I). Instead, similar to compiling into machine code for a register machine, we can *spill* excess values from registers to on-chip BlockRAM banks. We can employ the same infrastructure the compiler uses to manage these banks as scratch-pad memories to delay values for longer periods of time (Fig. 2), as long as these times are shorter than the initiation interval (II) of a loop. If that bound were exceeded, a value from the next iteration would overwrite one still needed in the current iteration. For many irregular computations (such as the complex solver codes examined here), the II of outer loops will often be very long (close the the total schedule length). This is actually amenable for a resource-shared microarchitecture, as the reuse of operators within the same iteration leaves them unavailable for computing data for the next iteration.

We operate the scratch pads in a simple dual-ported mode (one read and write in parallel). Thus, a single scratch pad can accept only a single value per stage, and also produces only a single value per stage. Since our datapaths need to delay many values, and some of them are read or written in parallel, spilling employs multiple scratch pads in parallel. We explicitly chose this approach over true dual ported operation, as in simple mode, we can give the logic synthesis tool the freedom to flexibly pick the best implementation (BlockRAM or LUT-based DistributedRAM) for each scratch pad.

Values are assigned to scratch pads by solving a graph coloring problem that assigns all accesses occurring in the same stage into separate banks. Our system can use the on-chip memory banks both for storing local arrays and to delay values, even mixing both uses within the same bank. New scratch pads are created and sized automatically as needed, they do not need to be predefined by the user.

4) Recomputation vs. Storage: The last approach we use to reduce NOP-chains is to *recompute* values as needed, instead of storing them. This can be done if, first, the inputs to the computation are already available in the correct stage *without* introducing the need to store *additional* intermediate values.

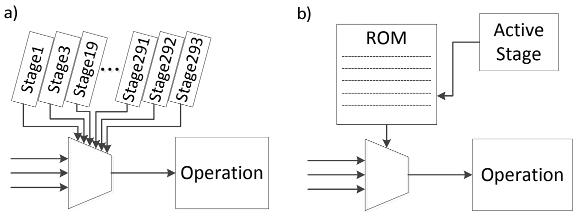

Fig. 3. Example for time-multiplexing operators

Already stored intermediate values do get reused here. Second, the required operators must be available (idle) in the target stage. If both conditions hold, the value is not stored but simply recomputed.

5) Combination of Methods: Nymble-RS exploits a combination of multiple of these methods to reduce area. As will be shown in Section V-B, the most efficient results are achieved by spilling longer lifetimes to scratch pads, and then using individual registers allocated using linear scan to provide the few remaining, very short NOP chains (spanning less than four stages).

B. Distributed and Partitioned Microcoded Controllers

The translation of complex irregular "C" code may result in a CDFG with thousands of stages, which would induce a correspondingly large number of FSM states in the controller. By itself, that would be manageable, as thousands of states are still efficiently mappable to one-flipflop-per-state controller implementations (note: not necessarily one-hot!). For a resource-shared datapath, however, the true difficulty (sketched in Fig. 3.a) lies in the extreme number of fan-ins each hardware operator, shared between multiple stages, would need to accept from the controller. Typically, these would be one-hot input multiplexer select signals from *each* associated controller state. This excessive number of signals is not efficiently implementable, even when the logic synthesis and mapping tools attempt to reduce routing congestion by logic/register replication (see Section V-A for a discussion).

As an alternative, we propose the use of *partitioned distributed microcode*. A binary-coded global micro-code program counter (μPCg) keeps track of the global execution state of the datapath. Each operators is provided locally with a dedicated micro-code ROM that controls their *individual* enable and select signals. By *distributing* the control signal sequences to the operators, the fan-in per-signal reduces to just one (the operator micro-code ROM, as shown in Fig. 3.b).

When this scheme is applied in its most basic form, each operator ROM would need to have a depth equal to the global number of stages. However, some operators are active only in parts of the schedule (see Section V for a discussion), most of the entries in their ROMs would zeros. As an alternative, we create shallower ROMs that just encompass the range between the first and last stages an operator is active (rounded up to the next power-of-2), which are addressed using *local* program counters μPCl. Each μPCl is started when the μPCg reaches the start of the operator's active stage range. Fig. 4 shows

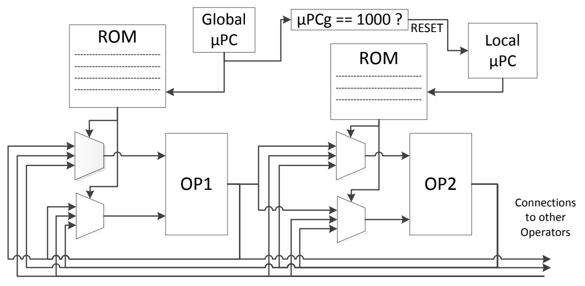

Fig. 4. Using a shallower microcode ROMs, locally addressed by μPCl

this for OP2, which is assumed to be active only in stages 1000. . . 1045 and thus more efficiently controlled by a 64-entry microcode ROM.

When switching from per-stage state-registers in the controllers to sequential microcode, we would normally lose the capability for pipelining. While this would not have a major impact on many of our use-cases, as heavily-resource shared micro-architectures have only limited potential for overlapped execution of iterations, we have devised a solution to lift this limitation: *Partitioning* the address spaces of the microcode ROMs leads to a μPC value of p activating the enable and select signals for both the stage p, as well as for the stage s where

$$(s < p) \quad \wedge \quad (s \ \mathbf{mod}\ \mathrm{II} = p \ \mathbf{mod}\ \mathrm{II}).$$

As an example, for II=600, an μPC=1011 would activate both stages 1011 and 411 simultaneously. Note that computations for the partitioned microcode address spaces are all performed at compile time and affect only the ROM contents, no additional hardware is required at execution time. For our benchmarks, this actually allows a small overlap of the execution of two iterations.

C. Faithful Rounding Floating Point Units

One way to further reduce the latency and required area of floating-point (FP) operators is the use of faithful rounding (FR) instead of IEEE754 Conforming Rounding (CR). FR can be less precise than CR: While CR requires the exact selection of the *single* representable value closest to the infinitely accurate result, FR relaxes this slightly: Here, an FP operator can *arbitrarily* return any one of the *two* closest representable values bracketing the accurate result. In literature, this rounding error is often expressed as a multiple of the "unit of least precision" (ULP). In CR, it cannot exceed 0.5 ULP, while FR has a maximum error of 1.0 ULP.

In Nymble-RS, we can selectively replace the Xilinx CoreGen FP multipliers and dividers by faithful rounding counterparts, as discussed in [20] and [21], respectively. To our knowledge, Nymble-RS is the first C-to-Hardware compiler offering this feature.

D. Schedule-dependent Tree-Height Optimization

Schedule-dependent tree-height optimization [22] has proven useful to reduce the height of trees of *identical* operations by taking into account the availability time of input signals. We have extended the technique beyond the usual arithmetic (e.g., add, multiply etc.) to also recognize specific *patterns* of comparators and multiplexers in the CDFG as **min** and **max** operations, which can then also be subjected to this optimization.

V. Experimental Evaluation

For evaluating Nymble-RS over spectrum of realistic input programs, we use CVXGEN to generate solver codes of increasing complexity. Four example problems provided on the CVXGEN website (SQP, SVM, Lasso, Portfolio) are translated with different problem dimensions and settings. In addition, Stan5p, a more complex example containing a complete model-predictive control problem for collision avoidance trajectory planning in autonomous ground vehicles, is used as fifth benchmark.

For the smaller examples, the following dimensions were entered into CVXGEN to create the solvers with static counts of 13K. . . 57K operations (mostly FP): SQP ($m = 3, n = 10 \to 13$ Kops), Lasso ($m = 100, n = 10 \to 57$ Kops), SVM ($N = 20, n = 4 \to 19$ Kops), Portfolio ($n = 25, m = 5 \to 30$ Kops). Stan5p has a total of 55 Kops.

```
void solve(void) {
  init();
  computeStartValue();
  while ( !solution_found ) {
    doOneIteration();
  }
}
```

Fig. 5. Algorithmic structure of the key function of the solver

Fig. 5 shows the pseudo-code of solver core. After initialization and computing a preliminary solution, the optimum is sought in an iterative manner. With the exception of Lasso, the while-loop requires more than 90% of the CPU computation time in all of the test cases. Even in Lasso, more than half of the execution time is spent there ($\approx$ 55%). Using the hardware-software co-synthesis capabilities of Nymble-RS, we mark only the compute intensive while-loop for hardware execution (saving chip area), leaving the remainder of the algorithm in software.

Two domain-specific optimizations are performed in addition to the standard LLVM passes: First, to enable the use of non-field sensitive alias analysis, the global structure containing the arrays the algorithm works with is decomposed into a global variable for each member, with all accesses being transformed correspondingly. Second, as the original code targets embedded CPUs, it tries hard to avoid divisions (see Fig. 6.a). Here, the condition test uses multiplication (cheaper) before performing the costly division. However, this code makes a parallel execution (by unrolling) or pipelining impossible. With the domain-specific knowledge that a[i] > 0, we transform this to the form in Fig. 6.b. Note that, for fair comparison, the other HLS tools in this study are also evaluated with this optimized code.

```
// (a) original , avoiding divisions
for (i=0; i<N; i++)
  if ( minval*a[i] < b[i] )
    minval = b[i] / a[i]

// (b) transformed into canonical min operation
for (i=0; i<N; i++)
  if ( minval < b[i] / a[i] )
    minval = b[i] / a[i]
```

Fig. 6. Transformation to canonical form of **min** operation

We target a Xilinx ZC706 board fitted with an XC7Z045 FPGA and use the "bare metal" software design flow. The generated hardware is accessed by the software using the AXI GP0 port, while the solver accelerator can access the main memory and second level cache using the AXI ACP port. Nymble-RS currently uses LLVM 3.3 as front-end and for machine-independent optimization, while Xilinx Vivado 2016.2 is used for logic synthesis and place&route. The initial evaluation uses a fixed microarchitecture of two FP adders, two FP multipliers, and one FP divider (Section V-E will explore other configurations). If not stated otherwise, the use of faithful rounding was enabled.

A. Microcode-based Controllers

Fig. 7 compares the resources required for the accelerators using conventional (dedicated state flip-flops) and microcoded controllers. The results are taken from the Vivado 2016.2 *post-synthesis* report, as the conventional controllers would not even fit on the Zynq device for the more complex solver examples. Interestingly, the older Vivado 2014.1, does much better on the conventional controllers than newer Vivado versions (which require almost 3x the LUTs). However, even comparing against the superior LUT numbers of the older Vivado version, the area required is still about an order of magnitude higher than that for our microcoded approach. On the other hand, the small increase of BRAM usage stays far below the limitations of the target device. Thus, microcode-based control is a key enabler for creating accelerators for complex irregular "C" code.

B. Handling of Intermediate Values

Table I examines the impact of the different intermediate result handling mechanisms discussed in Section IV-A. As baseline, we show a microcoded accelerator using just individual registers. Then we allow beneficial **recomp**utation of values instead of storing them. The **shift** column *additionally* enables the use of shift-registers for longer lifetimes, while the **spill** column *instead* combines selective recomputation with spilling values to scratch-pad memories. We show these impacts only for the two smaller examples SQP and Lasso, as even Nymble-RS will run out of memory on a 128 GB server when trying to process the larger solvers restricted to **base** mode (due to an excessive number of NOP operators). Again, the results given are *post-synthesis* areas. Note that spilling does also reduce the BRAM usage because reduced multiplexers require less rom, which more than compensates the rams

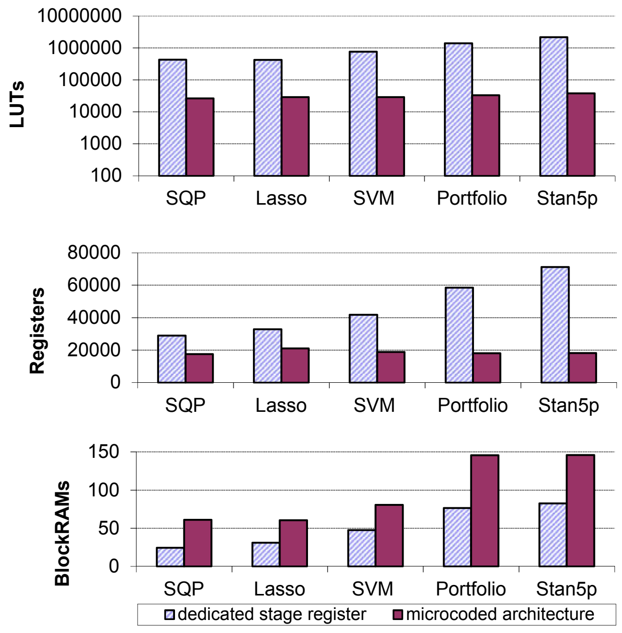

Fig. 7. Dedicated state registers vs. microcode: Resources required

		base	recomp	shift	spill
Registers	SQP	54512	37428	23552	17555
	Lasso	52566	32332	26964	20993
LUTs	SQP	254610	179300	51607	26653
	Lasso	224593	129857	48805	28960
BRAMs	SQP	581.5	306.5	69.5	61.0
	Lasso	511.0	196.5	64.5	60.5

TABLE I
POST-SYNTHESIS AREA REQUIREMENTS FOR DIFFERENT INTERMEDIATE VALUE HANDLING MECHANISMS

used as scratch pad. Here, 14 (SQP) and 17 (Lasso) scratch pads have been inserted with a size between 1 to 993 64-bit entries. Small scratch pads are automatically implemented in LUT-RAM instead of BRAM.

C. High-Level Performance Optimization

For a high-level look at performance, we initially consider the *schedule length* of the generated accelerators. Table II shows a significant reduction when using schedule-dependent min/max tree-height optimization and faithful rounding. The average schedule length reduction of min/max tree-height optimization is about 7.2%. Faithful rounding has an even higher impact, yielding an average reduction of 17.6%, with a maximum of 35.9% for Lasso.

D. Comparison to State-of-the-Art High-Level Synthesis Tools

For comparison with Nymble-RS, we have tried to synthesize the *smallest* test case SQP using a current industrial HLS system for Xilinx devices[1], and using the open-source LegUp compiler. Both tools support FP operations and pointers.

Earlier tests with LegUp 3.0 had failed due to its lack of a configurable upper limit for the number of floating point units.

[1]Anonymized due to licensing terms.

Benchmark	Schedule length FR off THO off	 FR off THO on	 FR on THO on
SQP	3366	3049	2079
Lasso	3790	3472	2216
SVM	3886	3491	3068
Portfolio	5931	5532	5289
Stan5p	8650	8218	7315

TABLE II
PERFORMANCE IMPACT OF FAITHFUL ROUNDING (FR) AND min/max TREE HEIGHT OPTIMIZATION (THO)

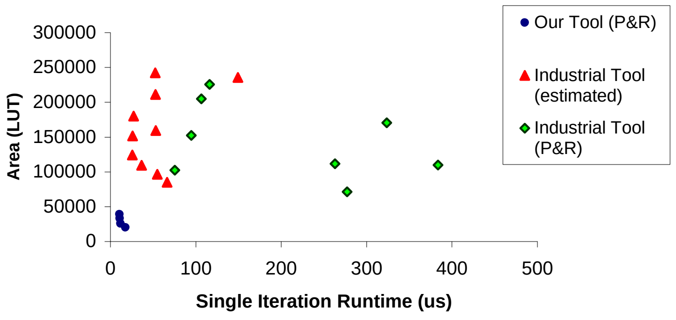

Fig. 8. Quality comparison of results for both compilers

While even LegUp 4.0, which does allow setting such a limit to force resource-sharing, still crashes in HW/SW co-compilation mode (called hybrid mode), it is able to progress further when using its pure hardware flow. Since LegUp does not fully support Xilinx devices yet, we have chosen to target the Altera Cyclone V family of chips for comparison. As LegUp interprets the constraints on FP units on a per-function basis, we set the limits to "one each of Mul/Add/Div", which (over the entire benchmark) still results in many more units being instantiated than with Nymble-RS (which applies the limits globally). To avoid this area explosion, we have experimented with inlining, using both the `static inline` keywords, as well as increasing the inlining threshold in LegUp's LLVM component. Even after this significant manual optimization effort, we could not achieve any result smaller than 122,737 LUTs and 186,929 registers for SQP. This is about 6x and 11x *larger* than the hardware generated by Nymble-RS given the same FP unit constraints. The LegUp-generated design did not fit even on the largest Cyclone V, so no maximum clock frequency can be given here. From simulation, we can determine that it would require 232K clock cycles, which is more than 7x of the slowest (=smallest, $<10\%$ of Z7045) SQP by Nymble-RS (31K cycles).

In 07/2016, the most recent version of the industrial tool was also able to translate the benchmark "C" code into hardware (earlier versions have failed). As with LegUp, we performed significant manual effort to tune the code for the tool: This included adding directives/pragmas for pipelining, loop unrolling, inlining, as well as constraining the microarchitecture to 1...2 FP units each and allowing numerically unsafe mathematical optimizations. Figure 8 compares the post-HLS estimates as well as post-P&R results of the industrial tool with the actual post-P&R results for Nymble-RS. We compute the runtime of a single iteration as the initiation interval (II) multiplied by the clock period (estimated and actual). Note the large discrepancy between the post-HLS estimates and the actual post-P&R delays for the industrial tool.

Despite $f_{\max}$ estimates of 140...164 MHz, no hardware could exceed 65 MHz on the target Z7045 SoC after P&R, with many results being unroutable or not even fitting on the Zynq device. For completeness, we also allowed the industrial tool to target a large Virtex 7 device, which resulted in post-P&R clock frequencies of 16...84 MHz. Here, results requiring fewer clock cycles often had difficulties reaching a high $f_{\max}$. Thus, even under optimal conditions (and a much more generous target device), the industrial tool's hardware is at least 3x larger and 7x slower than the results of Nymble-RS on the Z7045.

Remember that these experiments were performed on the *smallest* of the solver codes. Attempts to run the larger solvers through hardware synthesis either had failures in the flow, or resulted in *even larger* area growth and slowdown for LegUp and the industrial tool compared to Nymble-RS.

E. Design Space Exploration

Nymble-RS can flexibly scale the number of hardware operators used in the solver hardware. However, more operators do not always yield a correspondingly faster accelerator, as increasing the number of operators also leads to higher interconnect demand in the FPGA, which at some point will slow down $f_{\max}$. Thus, Nymble-RS can be used for design space exploration (DSE) to determine the best parallelism ./. frequency trade-off for each solver.

We show the impact of DSE for both the smallest SQP and the large Stan5p solvers in our benchmark suite in Table III. We report the wall-clock-time (WCT) for executing the *complete* application (hardware and software parts, as well as data transfers). When increasing the number of FP adders and multipliers, the schedules become shorter (due to exploiting more ILP), but $f_{\max}$ slows down slightly (due to interconnect). Especially for small solvers, using 4 (or more) has only small impact on the schedule length due to data dependencies.

F. Speed-Up vs. Software on Zynq SoC

Figure 9 finally compares the WCT of executing a single solve operation with and without hardware acceleration on the ZC706 platform. We compare the performance of our hardware accelerated solvers to pure software versions running on the 800 MHz ARM Cortex-A9 core. The software version of the solvers was compiled with -O3 and uses the hardwired NEON FPU in the processor. Following the recommendation of the CVXGEN authors, we also compiled the solvers with gcc -Os, and used the faster of the two builds for the comparison. In most cases, the hardware-accelerated solvers perform significantly better than their software-only counterparts. Note that the low system-level speed-up of Lasso is a benchmark-specific anomaly of the *software* code generated by HW/SW-co-synthesis not yet being optimal for the ARM processor.

Test Case	# Mul/Add	# FFs	# LUTs	# BRAMs	# DSPs	Schedule Length	II	$f_{\max}$ (MHz)	WCT (ms)
SQP	1/1	15824 (4%)	20503 (9%)	40.5 (7%)	25 (3%)	3519	3494	200.8	0.154
	2/2	17555 (4%)	26653 (12%)	61.0 (11%)	30 (3%)	2079	2054	178.0	0.113
	3/3	18905 (4%)	34024 (16%)	58.0 (11%)	35 (4%)	1876	1851	178.3	0.105
	4/4	20168 (5%)	39604 (18%)	59.5 (11%)	40 (4%)	1852	1827	177.9	**0.104**
Stan5p	2/2	18233 (4%)	38319 (18%)	146.0 (27%)	30 (3%)	7315	7290	178.2	0.440
	3/3	19415 (4%)	44505 (20%)	183.5 (34%)	35 (4%)	5749	5724	166.5	0.385
	4/4	20896 (5%)	58514 (27%)	251.5 (46%)	40 (4%)	5267	5242	157.5	**0.377**

TABLE III
DESIGN SPACE EXPLORATION. "WCT" IS WALL CLOCK TIME

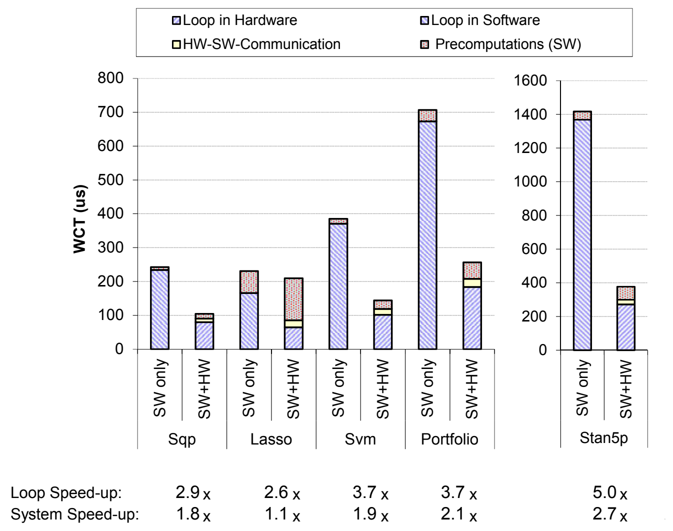

Fig. 9. Wall clock time on ZC706 platform with and without hardware acceleration

VI. CONCLUSION AND FUTURE WORK

Nymble-RS significantly exceeds the performance of state-of-the-art academic as well as industrial compilers for compiling floating-point-heavy irregular complex "C" code, even when expending significant manual effort of optimizing the code for the specific compiler, or allowing the competing tools to target larger devices. Furthermore, the performance of the accelerators created by Nymble-RS can be flexibly scaled, achieving significant speedups even when dedicating just a quarter of a mid-size FPGA to the accelerator circuit.

The speed-ups reported here are even more significant, as they are achieved not over relatively slow simple soft-core processors (often used as reference in related work), but against 800 MHz dual-issue out-of-order cores with hardwired FPU.

Future improvements to our flow will target the quality of the software code generated during hardware/software co-compilation, as well as consider the resource-shared use of advanced microarchitectures (e.g., datapath fusion [23]).

Acknowledgment: The authors would like to thank Xilinx Inc. for supporting their work by donations of hard- and software.

REFERENCES

[1] J. Huthmann, B. Liebig *et al.*, "Hardware/software co-compilation with the Nymble system," in *2013 8th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*. IEEE, Jul. 2013, pp. 1–8.

[2] M. Fingeroff and T. Bollaert, *High-Level Synthesis Blue Book*. Mentor Graphics Corp., 2010.

[3] Xilinx, Inc, *Vivado Design Suite User Guide – High-Level Synthesis*, 2012.

[4] Synopsys, Inc, *Synphony C Compiler User Guide*, 2011.

[5] A. Canis, J. Choi *et al.*, "LegUp: High-Level Synthesis for FPGA-based Processor/Accelerator Systems," in *Proc. Intl. Symp. on Field Programmable Gate Arrays (FPGA)*, 2011, pp. 33–36.

[6] J. Villarreal, A. Park *et al.*, "Designing modular hardware accelerators in c with roccc 2.0," in *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual Int. Symp. on*. IEEE, pp. 127–134.

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

[8] T. Callahan, "Automatic compilation of C for hybrid reconfigurable architectures," Ph.D. dissertation, UC Berkeley, 2002.

[9] S. Amarasinghe, J. Anderson *et al.*, "Multiprocessors from a software perspective," *Micro, IEEE*, vol. 16, no. 3, pp. 52–61, 1996.

[10] H. Lange, T. Wink *et al.*, "MARC II: A Parametrized Speculative Multi-Ported Memory Subsystem for Reconfigurable Computers," in *DATE'11*, 2011, pp. 1352–1357.

[11] J. Cong and W. Jiang, "Pattern-based behavior synthesis for FPGA resource reduction," in *Proceedings of the 16th Intl. ACM/SIGDA symposium on Field programmable gate arrays - FPGA '08*. New York, New York, USA: ACM Press, 2008, p. 107.

[12] J. Cong, H. Huang *et al.*, "A generalized control-flow-aware pattern recognition algorithm for behavioral synthesis," pp. 1255–1260, 2010.

[13] S. Hadjis, A. Canis *et al.*, "Impact of FPGA architecture on resource sharing in high-level synthesis," *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays - FPGA '12*.

[14] M. Benmohammed, S. Merniz *et al.*, "Asip micro-code generation from high-level specifications," in *Proceedings. 2004 Intl. Conf. on Information and Communication Technologies: From Theory to Applications*. IEEE, 2004, pp. 587–588.

[15] L. Boyd, Stephen and Vandenberghe, *Convex Optimization*. Cambridge university press, 2004, no. December.

[16] S. Grant, M and Boyd, "CVX: Matlab software for disciplined convex programming," 2008.

[17] J. Lofberg, "YALMIP : a toolbox for modeling and optimization in MATLAB," in *2004 IEEE Intl. Conf. on Robotics and Automation (IEEE Cat. No.04CH37508)*. IEEE, 2004, pp. 284–289.

[18] J. Mattingley and S. Boyd, "CVXGEN: a code generator for embedded convex optimization," *Optimization and Engineering*, vol. 13, no. 1, pp. 1–27, Nov. 2012.

[19] M. Poletto and V. Sarkar, "Linear scan register allocation," *ACM Transactions on Programming Languages and Systems*, vol. 21, no. 5, pp. 895–913, Sep. 1999.

[20] S. Banescu and F. D. Dinechin, "Multipliers for floating-point double precision and beyond on FPGAs," *ACM SIGARCH Computer ...*, 2011.

[21] B. Liebig and A. Koch, "Low-latency double-precision floating-point division for fpgas," in *Field-Programmable Technology (FPT), 2014 International Conference on*. IEEE, 2014, pp. 107–114.

[22] R. Hartley and A. Casavant, "Tree-height minimization in pipelined architectures," in *1989 IEEE Intl. Conf. on Computer-Aided Design. Digest of Technical Papers*. IEEE Comput. Soc. Press, pp. 112–115.

[23] M. Langhammer and T. VanCourt, "FPGA Floating Point Datapath Compiler," in *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*. IEEE, 2009, pp. 259–262.