From Software to Accelerators with LegUp High-Level Synthesis

Andrew Canis, Jongsok Choi, Blair Fort, Ruolong Lian, Qijing Huang, Nazanin Calagar, Marcel Gort, Jia Jun Qin, Mark Aldham, Tomasz Czajkowski, Stephen Brown, Jason Anderson

ECE Department, University of Toronto, Toronto, ON, Canada

legup@eecg.toronto.edu

Abstract

Embedded system designers can achieve energy and performance benefits by using dedicated hardware accelerators. However, implementing custom hardware accelerators for an application can be difficult and time intensive. LegUp is an open-source high-level synthesis framework that simplifies the hardware accelerator design process [8]. With LegUp, a designer can start from an embedded application running on a processor and incrementally migrate portions of the program to hardware accelerators implemented on an FPGA. The final application then executes on an automatically-generated software/hardware coprocessor system. This paper presents on overview of the LegUp design methodology and system architecture, and discusses ongoing work on profiling, hardware/software partitioning, hardware accelerator quality improvements, Pthreads/OpenMP support, visualization tools, and debugging support.

Categories and Subject Descriptors B.7 [*Integrated Circuits*]: Design Aids

Keywords High-Level Synthesis, Hardware Accelerators, FPGA

1. Introduction

Field-programmable gate arrays (FPGAs) are integrated circuits that can be programmed by the end user to implement *any* digital circuit. Since the dawn of the FPGA era in the 1980s, their size and complexity has tracked with Moore's Law, growing exponentially with each process generation. Today's largest FPGAs incorporate billions of transistors and can implement large complex systems. However, the growing complexity of such systems has made them increasingly difficult to design. One root of this problem is that most engineers are accustomed to software development—typically using `C`, which is considerably simpler than the circuit design and verification required for FPGA design.

Implementing a design on an FPGA can offer orders of magnitude improvement over a processor in terms of energy efficiency and performance for some applications [12, 25]. However, to access such benefits, an FPGA designer faces many challenges that do not exist in software development, such as choosing a suitable datapath architecture, implementing control logic, verifying the circuit functionality with a cycle-accurate simulator, and finally, using a static timing analysis tool to ensure timing constraints are met. The speed and energy efficiency of computing could be improved tremendously if this programmability hurdle could be lowered, especially considering that software developers outnumber hardware designers 10 to 1 [32].

One approach to ease the FPGA design burden is to use high-level synthesis (HLS), which automatically generates a cycle-accurate RTL circuit description from a high-level untimed `C` software specification. Recently, high-level synthesis has gained significant traction in industry as evidenced by many new commercial offerings: eXCite from Y Explorations [34], Calypto Catapult [16], Forte Cynthesizer [19], Xilinx Vivado [21], and Altera's OpenCL Compiler [17]. The advantage of high-level synthesis is that a circuit designer can work more productively at a higher level of abstraction, and achieve faster time-to-market than using manually designed RTL.

We have implemented an open-source high-level synthesis research framework called *LegUp* [8]. An overarching goal of LegUp is to offer a programming paradigm for FPGAs that simplifies the design process for software engineers who are not familiar with hardware design. Using LegUp, the designer first implements their algorithm on a processor, then incrementally offloads portions of the program into hardware accelerators that execute in tandem with the processor, thereby achieving improved speed and energy consumption. The tool encourages the exploration of the software/hardware design space, for example, by running only critical or highly parallelizable portions of the application in hardware, and running the remainder on a processor. Being one of the few robust open-source HLS frameworks, LegUp is a powerful platform that enables research in a variety of areas including HLS algorithms, hardware/software co-design, and embedded systems. LegUp 3.0 was released in January 2013 and is available for download at: `http://legup.eecg.toronto.edu`.

In this paper, we provide an overview of the LegUp project and discuss our recent research directions. The remainder of this paper is organized as follows: Section 2 provides an overview of LegUp's design methodology and target architecture. Section 3 presents a method of automating the hardware/software partitioning when retargeting an algorithm from a software implementation to an FPGA processor/accelerator system. Section 4 discusses quality improvements that we have made to LegUp's high-level synthesis algorithms. Section 5 presents LegUp's support for parallel programs described with Pthreads and OpenMP, which can use multiple hardware accelerators running concurrently. Section 6 discusses work-in-progress on a debugging platform and also describes our hardware visualization tool. Conclusions are given in Section 7.

2. LegUp Flow Overview

Fig. 1 illustrates the steps of the LegUp design methodology. Referring to the labels in the figure, at step ①, the designer implements their application in software using `C`, targeting a soft-core MIPS processor running on the FPGA [33]. As the application executes in step ②, a built-in hardware profiler [2] identifies critical sections of the code that would benefit from a hardware implementation. Using this profiling information in step ③, the user marks functions in the program to be synthesized into hardware accelerators. The application is re-compiled by LegUp in step ④, which automatically converts the marked sections into hardware acceler-

978-1-4799-1400-5/13/$31.00

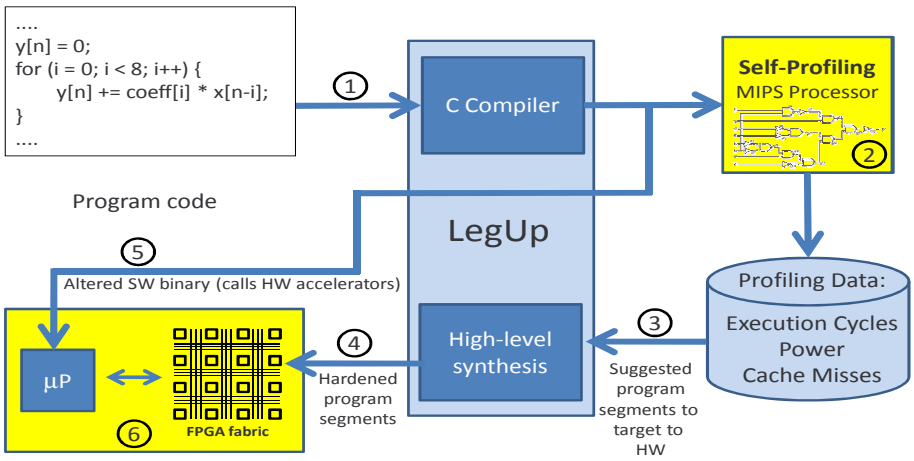


Figure 1. Design flow with LegUp.

ators using LegUp's high-level synthesis engine. Next, in step ⑤, the original software is re-compiled with the accelerated functions replaced with code to start the corresponding hardware accelerators and pass any required function parameters to the hardware. Finally, the hybrid processor/accelerator system executes on the FPGA in step ⑥. In this self-accelerating adaptive system, the designer can harness the performance and energy benefits of an FPGA using an incremental methodology, while limiting time spent on hardware design. The LegUp programming flow bears some similarity to GPGPU programming (using languages like CUDA [26] and OpenCL [1]) in the sense that we allow the programmer to iteratively and incrementally work to raise performance, with the whole program working correctly at all times.

2.1 LegUp System Architecture

LegUp can target two Altera FPGAs: the Cyclone II on the Altera DE2 board [5], and the Stratix IV on the Altera DE4 board [6]. The target system architecture is shown in Fig. 2. The system comprises the MIPS soft processor, hardware accelerators, on-chip cache, as well as off-chip memory (8MB SDRAM on the DE2 board or 2GB DDR2-SDRAM on the DE4 board). An accelerator may have local memories for storing data that is not shared with the processor or other accelerators. These local memories are implemented in on-chip block RAMs, instantiated within a hardware accelerator. Data shared between the processor and hardware accelerators is stored in off-chip memory, which can be accessed using the on-chip cache. The components of the system communicate via the Avalon Interconnect, Altera's on-chip interface, which is generated automatically by Altera's SOPC Builder tool [7]. Avalon is a point-to-point network, which allows multiple independent transfers to occur simultaneously via memory-mapped addresses. When multiple components are connected to a single component, such as the on-chip data cache, a round-robin arbiter is generated to arbitrate among simultaneous accesses.

2.2 Multi-ported caches

When many accelerators are operating in parallel, memory bandwidth can easily become a performance bottleneck. The on-chip RAMs on current commercial FPGAs have two ports, meaning that for a given memory block, there can only be up to two memory accesses at a time. However, for systems with many accelerators that need to access memory concurrently, two ports may not be adequate and cache accesses may limit performance. A typical way

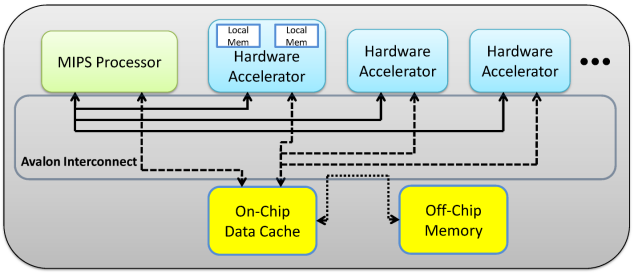


Figure 2. Default LegUp system architecture.

to increase memory bandwidth is to use multiple coherent memory blocks, with extra circuitry to manage memory coherency between the memory blocks. However, by implementing memory coherency we add area and latency overhead. Thus, we take an alternate approach, where we implement multi-ported memories (that have more than 2 ports) using existing dual-ported memory blocks. We can then use these multi-ported memories to implement multi-ported caches suitable for many-accelerator systems.

We have investigated two types of multi-ported caches, called the *LVT* cache and the *MP* cache [10], both of which allow multiple concurrent accesses to all regions of the cache in every clock cycle. The *LVT* cache is based on memory replication, whereas the *MP* cache uses memory multi-pumping (operating the memory at a higher clock rate than the surrounding system). The main advantage of both cache architectures is that they offer higher on-chip memory bandwidth than what is typically available on the FPGA fabric, while providing a shared memory space which acts as a *single* piece of memory. These caches also require no cache coherency scheme, avoiding the area and latency costs for synchronization.

3. Hardware/Software Partitioning

With the LegUp design methodology, the program is partitioned into both a hardware portion and a software portion. The chosen partitioning depends on the designer's objective, which is often to reduce overall execution time. Towards this goal, the MIPS soft processor contains a hardware profiler to determine which sections of the original program are taking the most execution time. LegUp can also *estimate* the speedup associated with migrating a particular program segment into hardware versus leaving it in software.

3.1 Hardware Profiling

The hardware profiler in the MIPS soft processor is called LEAP, which stands for Low-overhead and Extensible Architecture for Profiling [2]. For each function in a program, the profiler can be used to quickly and accurately obtain the exact number of clock cycles spent executing the function. In software-based profiling, the program being profiled must be modified with instrumentation to gather profiling data during its execution. In contrast, our hardware-based approach allows the program to execute in its original unmodified form at full speed on the processor. The MIPS processor is augmented with additional circuitry that automatically gathers profiling data as the program executes. Such hardware profiling is superior in speed and accuracy when compared to software profiling.

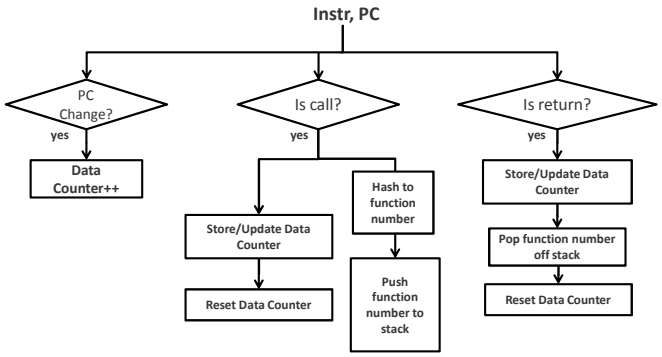


Figure 3. High-level flow chart for instruction-count profiling.

The high-level operation of LEAP is shown in Fig. 3. LEAP profiles the execution of the program by monitoring the processor's program counter and instruction bus. During execution, LEAP maintains a counter, called a *Data Counter*, that tracks the number of times an *event* has occurred. Two modes are available: the profiler can count dynamic instructions, or clock cycles.

LEAP organizes the collected data on a per-function basis by allocating a storage counter for each software function. LEAP identifies function boundaries by decoding (in hardware) the executing instruction to determine if it is a function call or return. If a call is detected, the Data Counter is added to any previously stored values associated with the function containing the call instruction (from previous invocations of the function). The Data Counter is then reset to 0 to begin counting the events in the called function. If a function return is detected, the Data Counter value is added to the counter associated with the current function, and once again the Data Counter is reset.

In order to determine the counter associated with a particular function, other hardware profilers, such as SnoopP [30] (a hardware profiler for FPGA-based processors), use a large number of comparators to associate program counter address ranges with individual counters. A novel aspect of LEAP is the use of perfect hashing hardware to associate function addresses with counters. A set of hashing parameters are generated during the software compilation stage (step ① in Fig. 1) and used to configure the profiler on the FPGA. No modifications of the hardware profiler circuit (e.g. resynthesis or reprogramming) are needed to profile a new program. The use of hashing leads to significantly less hardware overhead when compared to other hardware profilers. Specifically, relative to SnoopP, our design requires up to 18× less area [2].

3.2 Accelerator Speedup Prediction

By using the LEAP profiler, the user can identify time-consuming program segments. However, these compute-intensive functions may not be suitable for hardware acceleration, perhaps because they contain a sequential algorithm with minimal instruction level parallelism or they are too memory intensive. Ideally, we would know exactly how much execution time would be saved by synthesizing a segment of software code into a hardware accelerator.

One way we could gauge the speedup achieved by hardware acceleration is to actually convert the segments into hardware circuits and then run the program on the board to measure the results. But this approach is too time consuming if there are many alternatives to investigate, as it requires running FPGA synthesis and place-and-route tools for each alternative. Alternately, one could run an RTL simulation to measure the execution time, in cycles, of the final hybrid system. However, this method is also time-consuming and becomes infeasible in real applications.

To aid in the task of software/hardware partitioning, LegUp provides an *estimate* of the total number of clock cycles consumed by a function if it is accelerated in hardware, which can then be compared to the LEAP profiling results described above. This approach uses profiling (in software) to estimate the execution flow of the processor/accelerator hybrid system and then uses early high-level synthesis scheduling information to predict the number of cycles required by portions of the program after being synthesized to hardware.

When synthesizing a software program into a hybrid system, LegUp replaces the accelerating functions with wrapper functions to enable communication between the processor and accelerators. The call to a wrapper function starts the accelerator's execution and the return from the wrapper function indicates the accelerator finished its execution. Our estimation approach considers the hardware cycles spent on three operations: 1) the execution of the hardware accelerator, 2) the accelerator's initialization performed by the software wrapper function, and 3) reads and writes to the shared memory space.

We estimate the cycles taken during the accelerator's execution in two steps. First, we perform HLS scheduling for the accelerated function to determine the number of clock cycles required for each basic block in the function. A basic block is a contiguous set of instructions with a single entry (at its beginning) and exit point (at its end). Next, we execute the program in software using representative inputs to estimate the number of times each basic block is executed. Finally, we estimate the total cycle count of the accelerated function by multiplying the estimated number of times each basic block is executed by the number of clock cycles required by the corresponding basic block in its schedule.

To estimate the time taken by the software wrapper function running on the processor, we count the number of instructions required by the wrapper. The instruction count is sufficient for wrapper function estimation, as the wrapper function is small and only contains simple operations to communicate with hardware accelerators, and we have found empirically that the instructions-per-cycle of the MIPS processor is close to one.

We estimate the cycles spent accessing shared memory in three steps. First, we run the program with a representative set of inputs using a MIPS emulator to determine the address sequence accessed by the software program (without hardware accelerators). Next, we predict the address sequence accessed by the hybrid processor/accelerator system by eliminating any addresses that are stored in local memory of the hardware accelerators. Then, we use a cache simulator to determine the number of cache hits and misses. Finally, we can use the estimated cost of a cache hit or miss (in cycles) to predict the total cycles spent on shared memory accesses.

Experimental results show that our approach has an average error rate of about 7% compared to the results obtained from RTL simulation, but with 184× less run-time on average.

3.3 Partitioning Example

An example of hardware/software partitioning is provided in Table 1 for four functions of the `jpeg` benchmark in the CHStone benchmark suite [15]. In the "Profiling" column, the table presents the number of cycles required by the function when running on the MIPS soft processor (profiled using LEAP). In the "Estimation" column an estimate of the execution cycles are given when the function is migrated to a hardware accelerator. The "Simulation" column gives the actual number of execution cycles measured using cycle-accurate simulation of the generated hybrid system. As shown in the table, if we select the function with the most software cycles (on the MIPS determined using LEAP) for acceleration, function *buf_getb* will be synthesized to hardware, resulting in 63,337 cycles of actual reduction versus a pure-software implementation. However, by choosing the function that has the highest estimated cycle reduction, function *YuvToRgb* will be synthesized to hardware, leading to an execution time reduction of 327,308 cycles.

Currently, based on the profiling and estimation results, users can choose which function to accelerate based on the estimated speedups. In the long run, we would like LegUp to act as a *self-accelerating adaptive processor*, that will profile running applications and automatically synthesize code segments into hardware, improving performance without user intervention.

Table 1. Software Cycles and Hybrid Cycles of *JPEG* Benchmark.

	Profiling	Estimation		Simulation	
Functions	Software Cycles	Hybrid Cycles	Reduced Cycles	Hybrid Cycles	Reduced Cycles
buf_getb	713,102	582,386	130,716	649,765	63,337
ChenIDct	657,775	678,617	-20,842	681,366	-23,591
YuvToRgb	569,501	158,891	410,610	242,193	327,308
buf_getv	564,650	222,027	342,623	241,306	323,344

4. Hardware Accelerator Quality

High-level synthesis has traditionally been divided into three steps [13]: allocation, scheduling and binding. Allocation determines the properties of the target hardware: the number of functional units available, the number of pipeline stages of each functional unit, and the estimated functional unit delay. Scheduling assigns each operation to a state, while satisfying data and control dependencies, and constructs a finite state machine to control the datapath. The LegUp HLS tool uses SDC scheduling [11], which formulates the scheduling problem mathematically as a linear program. Binding is performed after scheduling to assign the operations in the program to hardware functional units. When multiple operators are assigned to the same hardware unit, multiplexers are added to facilitate the sharing. LegUp uses a weighted bipartite matching heuristic to solve the binding problem [22], which can be optimally solved in polynomial time [24].

An ongoing challenge in high-level synthesis is to generate circuits that can meet realistic FPGA design constraints that are comparable to hand-designed digital circuits. Towards this goal, we have improved the high-level synthesis algorithms of LegUp by analyzing the impact of compiler passes, investigated the impact of bitwidth minimization, added support for loop pipelining and explored multi-pumping the FPGA DSP blocks.

4.1 Compiler Passes

Modern HLS tools are implemented within software compiler frameworks, and consequently, the programs that are input to such tools are subjected to standard compiler optimizations applied before HLS commences. Compilers perform their optimizations in *passes*, where each pass is responsible for a specific code transformation, for example, dead-code elimination, constant propagation, loop unrolling, or loop rotation. The passes within these compiler frameworks were intended to optimize software programs that run on a microprocessor. We studied the impact of these passes for HLS-generated hardware.

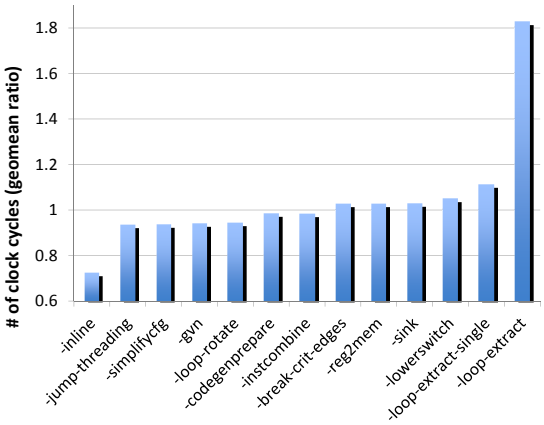


Figure 4. Impact of individual compiler passes on geomean clock cycle latencies across 11 benchmarks.

LegUp is implemented within the LLVM compiler framework [20], which contains implementations for 56 optimization (transform) passes that may alter the program. LLVM represents the program being compiled using an intermediate presentation (IR) that is essentially machine-independent assembly code. Compiler passes receive the IR as input, and produce an optimized IR as output. The familiar command-line optimization levels (e.g. `-O3`) correspond to a particular set and sequence of compiler passes. While the standard compiler optimization levels offer a simple set of choices for a developer, the particular optimizations applied at each level are generally chosen to benefit the run-time of a basket of programs. It is not guaranteed, for example, that for a specific program the `-O3` level produces superior results to the `-O2` level. This has led the (software) compiler community to consider selecting a particular set of compiler optimization passes on a per-program (or even per-code-segment) basis. Such "adaptive" compiler optimization has been the subject of active research in recent years, with a few examples of highly-cited works being [3, 27, 31].

Our research [23] in this area has focussed on two issues: 1) determining the impact of different compiler passes on HLS-generated hardware and 2) creating an HLS-oriented approach to the application of compiler optimization passes.

4.2 Analysis of Passes

To understand the effect of compiler passes on hardware, we conducted a wide range of experiments to explore: 1) the impact of each LLVM pass in isolation, and 2) the impact of pass ordering.

We begin by analyzing LLVM optimization passes in isolation relative to `-O0` (no optimization). Fig. 4 shows how a subset of passes individually affect the number of hardware execution cycles – the *cycle latency*. The horizontal axis lists the names of each pass. The vertical axis represents the geometric mean ratio (over 11 benchmarks) of cycle latency when a particular pass is used, relative to the `-O0` case. Values less than 1 represent reductions in cycle latency relative to the baseline case. Of the 56 different LLVM passes, only the 13 passes shown in the figure, impacted the geomean cycle latency by more than 1% when applied in isolation. Observe in the figure that passes, `-loop-extract` and `-loop-extract-single` caused a large increase in the geomean number of execution cycles. Both of these optimizations ex-

tract loops into separate functions. The LegUp HLS tool does not optimize across function boundaries, and moreover, implements each function as a separate Verilog module, with handshaking between modules occurring when one function calls another. Outlining loops as functions therefore naturally leads to higher numbers of execution cycles. The `-inline` pass has precisely the opposite effect: a large decrease in cycle latency is observed when callees are collapsed (inlined) into callers.

Another observation obtained by analyzing passes in isolation, was that the set of beneficial passes is highly benchmark dependant. Therefore, we created custom "recipes" of passes tailored to each benchmark. Only those passes that positively benefited the particular benchmark were selected and were ordered alphabetically. The results of these recipes showed clock cycle latency improvements over `-O3` (the default optimization level for LegUp) for 10 of our 11 benchmarks.

We also considered the order in which passes are applied. We selected 33 passes, comprised of all those passes that had an impact in isolation (on top of `-O0`) and also those passes that had an impact when removed from `-O3`. We considered all pairs of passes (of which there were $\binom{33}{2} = 528$ pairs) from this group and evaluated the pairs in both orders. Of the 528 pass pairs, 411 had a difference in their impact depending on the ordering. The results clearly demonstrate the importance of pass ordering on HLS quality of results for the majority of pass pairs.

4.3 HLS-Directed Compiler Optimizations

Given our experience with customized recipes and the observation that the compiler passes beneficial to each benchmark are both benchmark dependent and order dependent, we felt it would be difficult to devise a single recipe of passes that would benefit *all* circuits. We therefore proposed a HLS-directed approach to the application of compiler optimization passes. At a high level, our approach works as follows: we iteratively apply one or more passes and then "score" the result by invoking partial HLS coupled with rapid profiling (in software). Transformations made by passes deemed to positively impact hardware are accepted. Conversely, we *undo* the transformations of passes that we predict to be damaging to hardware quality.

We implemented and evaluated three variants of our HLS-directed approach to the application of passes, which we refer to as the *iteration method*, the *insertion method*, and the *insertion-3 method*. The first two variants differ from one another in their implementation of how a chosen pass p are applied to the best IR found so far. In the iteration method, we traverse all passes in an order based on the pairs analysis results (as described in Section 4.2) so that the pairwise pass ordering favors reductions in clock cycle latency. We apply the passes in order, in particular, we apply the selected pass, p, at the *end* of the pass recipe that produces the best IR so far.

In the insertion method, we consider all possible insertion positions for p in the recipe that produced the best IR so far, and keep the recipe and IR corresponding to the insertion position that produced the IR with the lowest number of clock cycles. Our last variant, insertion-3, extends the insertion method by storing the top 3 IRs and recipes, instead of storing the single best IR and recipe. In insertion-3, the chosen pass p is applied to all 3 of the top IRs/recipes.

Table 2 shows the geomean and ratio of speed-performance results over 11 benchmark circuits optimized using five different compiler optimization flows: no optimization (`-O0`), standard `-O3` optimization, the iteration method, insertion method, and the insertion-3 method. First, observe that `-O3` provides a clear advantage over `-O0`: clock cycle latencies without any optimization are 12% higher, on average, vs. with `-O3`. All of the flows produce significantly better results than `-O3`, on average. The iteration method provides 10% improvement; the insertion method offers 11% improvement; and, the insertion-3 method provides 17% improvement in cycle latency. These results tracked very well with the improvement in wall clock time, as shown in right-side of the table.

Table 2. Compiler passes performance results (IT: Iteration Method, IN: Insertion Method, IN3: Insertion-3 Method).

	Clock Cycles		Wall Time (μs)	
Flows	**Geomean**	**Ratio**	**Geomean**	**Ratio**
-O0	18,404	1.12	300	1.16
-O3	16,381	1.00	260	1.00
IT	14,717	0.90	231	0.89
IN	14,572	0.89	229	0.88
IN3	13,641	0.83	217	0.84

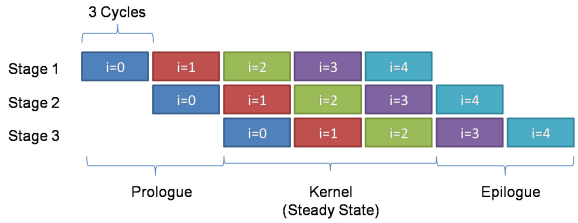


Figure 5. Time sequence of a loop pipeline with II=3 and five loop iterations.

We believe the automated approaches to selecting compiler optimizations on a per-program basis are practical, and will be of keen interest to FPGA users seeking high design performance. Such approaches also appear to be a useful mechanism for narrowing the gap between HLS-generated hardware and manually-designed RTL.

4.4 Loop Pipelining

In many applications, the majority of time is spent executing critical loops. Loop pipelining is a way of extracting parallelism automatically from a program by analyzing loops and generating hardware pipelines to exploit the inherent parallelism across loop iterations. Loop pipelining is based on a compiler technique traditionally aimed at VLIW processors called software pipelining. A popular software pipelining technique is called *iterative modulo scheduling* [28], which has been adapted for loop pipelining in high-level synthesis by C-to-Verilog [18], PICO [29], and also by LegUp.

Iterative modulo scheduling combines list-scheduling, backtracking, and a resource reservation table to reorder instructions from multiple loop iterations into a set of stages comprising the loop *kernel*. The kernel of a loop pipeline starts a new loop iteration every *II* cycles, where *II* is the *initiation interval*, which is also the number of cycles between successive inputs to the pipeline. The kernel consists of one or more pipeline stages that all execute in parallel.

Fig. 5 shows the time sequence of a loop pipeline with an initiation interval of three cycles and a kernel consisting of three pipeline stages executing in parallel. At any time step in the steady-state operation of the pipeline, we are executing operations from three consecutive iterations of the loop, one in each pipeline stage. For instance, when the pipeline initially reaches steady state, loop iterations: $i = 0$, $i = 1$, and $i = 2$ are all executing, and iteration $i = 0$ is finishing. In Fig. 5, the prologue is the time period when the pipeline is filling up, while the epilogue is the time period when the pipeline is flushing (loop execution is concluding). Compared to sequential execution, loop pipelining increases parallelism by

overlapping the execution of loop iterations, which decreases the time required to complete the loop while increasing hardware utilization.

LegUp supports loop pipelining of simple loops which consist of a single basic block and where the loop bounds are not modified during loop execution. The loop body can contain multi-cycle operations such as floating point and memory operations. Simple cross-iteration dependencies are supported with conservative alias analysis. The current implementation assumes that whenever the loop body contains a read and write to the same array that a dependency exists between the current and previous loop iteration. Future releases of LegUp will include more advanced cross-iteration dependency analysis.

4.5 Multi-Pumping

For applications that involve many multiplication operations, LegUp uses a new approach to resource sharing that allows multiple operations to be performed by a single multiply functional unit in one clock cycle [9]. Our approach is based on multi-pumping, which operates functional units at a higher frequency than the surrounding system logic, typically $2\times$, allowing multiple computations to complete in a single system cycle. This method is particularly effective for the DSP blocks on modern FPGAs. The hardened DSP blocks in modern FPGAs can operate a speeds exceeding 500 MHz, whereas typical system speeds are less than 300 MHz. We have found that multi-pumping is a viable approach to achieve the area reductions of resource sharing, with considerably less negative impact to circuit performance. For a given constraint on the number of DSPs, multi-pumping can deliver considerably higher performance than resource sharing. Empirical results over digital signal processing benchmarks show that multi-pumping achieves the same DSP reduction as resource sharing, but with a lower impact to circuit performance: decreasing circuit speed by only 5% instead of 80%.

4.6 Bitwidth Minimization

Software programs today use standard datatypes that are 8, 16, 32, or 64-bits in length. As such, programs are *over engineered* in the sense that variables are frequently represented using more bits than are actually required, e.g. a 32-bit `int` datatype may be used for a loop index that is known to have a range from `0` to `100`. Because processor datapaths are of fixed widths, there is little to be gained in term's a software program's performance by optimizing bitwidths. However, in HLS, hardware quality (area, speed and power) is impacted considerably by the bit-level representation of program variables.

LegUp uses two strategies to statically (i.e. at compile time) or dynamically (i.e. using run-time profiling) determine minimized representations of variables: 1) range analysis and 2) bitmask analysis. Range analysis seeks to determine the maximum and minimum values that variables take on in a program's execution and in so doing, bound the number of bits required to represent the variable. Variable ranges can be deduced from constants in the source code, and then propagated through a program's control-dataflow graph to infer ranges for other variables. Bitmask analysis, on the other hand, seeks to characterize the individual bits in a variable. For example, assume that A and B are unknown 16-bit values and consider the C-language statement: `Z = A & (B << 2)`. In this case, the two right-most bits of Z are guaranteed to be logic-0 and this property can be applied to minimize the size of hardware that uses Z as an operand (e.g. if Z feeds into a multiplier, the two right-most bits of the product are guaranteed to be logic-0). Note that while *bitmask* analysis guarantees that Z's two LSBs are 0, *range* analysis can infer nothing regarding Z's min and max values. The two forms of analysis thus offer complementary information.

Table 3 shows the results of applying bitwidth minimization techniques for the set of CHStone [15] benchmarks, targeted to the Altera Cyclone II $90nm$ commercial FPGA [4]. Based on static analysis alone that analyzes both ranges and bitmasks, circuit area can be reduced by 9%, on average, compared with Altera's Quartus II RTL synthesis tool, which itself significantly prunes the circuit based on constants in the RTL code. With additional dynamic profile-driven analysis, area reductions increase to 34%, on average, with the caveat that results are only guaranteed to be correct if the values of variables remain within the ranges that were observed during profile-driven analysis. Full details of our bitwidth minimization approach can be found in [14].

5. Pthreads and OpenMP

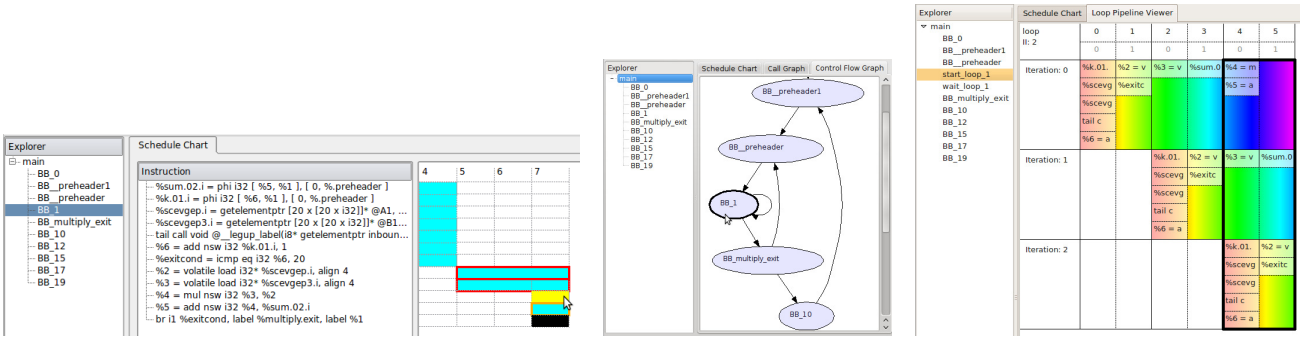
One source of the quality gap between HLS-generated hardware and human-designed hardware is the inability of HLS to fully exploit the parallelism available in the target FPGA fabric for a given application. Current HLS tools can typically employ instruction level parallelism and loop pipelining to execute multiple operations in parallel. This fine-grained parallelism, however, is often not enough to meet the performance requirements of a high-performance system. Coarse-grained parallelism is often realized by using an HLS tool to synthesize a single hardware core, and then manually instantiating multiple instances of the core in structural HDL. Some commercial HLS tools, such as Vivado [21], allow this to be done through vendor-specific pragmas. Although the use of vendor-specific pragmas can ease the process of instantiating multiple hardware cores, it nevertheless requires knowledge of hardware design – a barrier for software engineers. We addresses this challenge by providing a mechanism through which an engineer may use software techniques to specify parallelism to the LegUp HLS tool, with the tool then implementing the specified parallelism in a hardware circuit.

LegUp provides support for two standard parallel programming methodologies which software engineers are likely familiar with – Pthreads and OpenMP. Parallelism described in the software code is automatically synthesized into parallel hardware accelerators that perform the corresponding computations concurrently. Parallel programming in software often requires the use of synchronization constructs that, for example, manage which threads may execute a given code segment at any given moment. Recognizing this, we also provide HLS support for two key thread synchronization constructs in the Pthreads/OpenMP library: mutexes and barriers. The approach we take is to automatically instantiate parallel hardware for parallel threads. That is, each software thread is mapped automatically into a hardware accelerator. The remaining (sequential) portions of the program are executed in software on the MIPS soft processor.

Table 4 shows a list of Pthreads and OpenMP library functions which are currently supported by LegUp. In addition to those listed in the table, OpenMP clauses to set the number of threads (`num_threads`), the scopes of variables (e.g. `public`, `private`) and the division of work among threads (`static` scheduling of any chunk size) are also supported. Note that all of the OpenMP/Pthreads functions in Table 4 are automatically compiled in our framework, requiring no manual code changes by the user. Meaning that, the input `C` program with calls to the Pthreads/OpenMP API can be compiled to a hybrid processor/accelerator system *as is*. The complete system, including the MIPS processor, on-chip cache, off-chip memory controller, as well as parallel accelerators, can be created with a *single* make target.

Table 3. Bitwidth minimization Cyclone II implementation results.

Benchmark	LUTs			Registers			FMax (MHz)		
	Baseline	Bitmask+ Range	Dynamic+ Bitmask	Baseline	Bitmask+ Range	Dynamic+ Bitmask	Baseline	Bitmask+ Range	Dynamic+ Bitmask
dhrystone	5244	4120	3738	3575	3131	2438	117.94	114.09	115.96
fft	2046	2043	1880	1048	1028	746	92.89	91.3	91.3
adpcm	21695	18631	7036	11039	10020	4291	55.46	56.04	56.16
aes	19784	15792	8871	11470	9162	4066	49.38	49.82	46.47
blowfish	10621	10590	10296	7412	7353	7040	75.41	73.61	71.62
gsm	9787	9645	7807	6612	6487	5029	33.2	32.39	32.98
jpeg	33618	31083	22057	20688	19388	11885	18.02	17.53	19.15
mips	3384	3358	2116	1620	1590	999	98.8	95.56	110.22
motion	4054	4020	2946	2526	2526	1656	112.18	111.83	125.85
sha	10686	8243	7612	7779	5838	5371	99.42	106.68	109.42
Geomean:	**8655**	**7838**	**5711**	**5230**	**4794**	**3217**	**65.7**	**65.2**	**67.3**
Ratio:	**1.00**	**0.91**	**0.66**	**1.00**	**0.92**	**0.62**	**1.00**	**0.99**	**1.02**



(a) Schedule Gantt Chart (b) Control Flow Graph (c) Loop Pipeline Schedule

Figure 6. Screenshots of the LegUp visualization tool.

Table 4. Supported Pthreads functions/OpenMP pragmas.

Pthreads Functions	Description
pthread_create(..)	Invoke thread
pthread_join(..)	Wait for thread to finish
pthread_exit(..)	Exit from thread, can be used to return data
pthread_mutex_lock(..)	Lock mutex
pthread_mutex_unlock(..)	Unlock mutex
pthread_barrier_init(..)	Initialize barrier
pthread_barrier_wait(..)	Synchronize on barrier object
OpenMP Pragmas	**Description**
omp parallel	Parallel section
omp parallel for	Parallel for loop
omp master	Parallel section executed by master thread only
omp critical	Critical section
omp atomic	Atomic section
reduction(operation: var)	Reduce a var with operation
OpenMP Functions	**Description**
omp_get_num_threads()	Get number of threads
omp_get_thread_num()	Get thread ID

6. Visualization and Debugging

LegUp provides visualization tools for analyzing the internal HLS algorithms. For instance, we have a graphical viewer for the scheduling report file produced by LegUp that shows a Gantt chart of the scheduled instructions for the program and also can visualize loop pipeline scheduling. Fig. 6 shows three screenshots of the LegUp visualization tool for a matrix multiply kernel. Fig. 6a shows a Gantt chart for LegUp's high-level synthesis schedule. On the left side, the "Explorer" panel lists each basic block for each function, in this case the user has selected the basic block labeled "BB_1". In the "Schedule Chart" window pane the schedule viewer gives a list of all LLVM instructions inside the selected basic block. Each LLVM instruction corresponds to a hardware operation in the synthesized circuit. The user can highlight any instruction to display the data dependencies between all predecessor and successor instructions. Fig. 6b shows the control flow graph for the kernel, where each node in the graph is a basic block. Fig. 6c shows the loop pipeline schedule after the basic block has been pipelined. The pipeline initiation interval is two, which means a new loop iteration begins every two clock cycles. The area highlighted in black is the steady-state operation of the pipeline; observe that three iterations of the loop are executing in parallel.

In addition to visualization, we have been focusing recently on adding debugging capabilities to LegUp. Debugging tools are ubiquitous in the software development community because they raise productivity by providing insight into the execution state as a program executes. In contrast, most hardware designers are accustomed to using simulation waveforms to debug their digital circuits. With LegUp, we want to bridge this gap by offering users a software-like debugging platform for the hybrid hardware/software coprocessor system. LegUp's debugging platform will help developers gain insight into problems with their applications at a higher level of abstraction than traditional RTL simulation and waveform analysis.

To implement the debugger, LegUp leverages the LLVM compiler debugging meta-data, which maps each `C` statement to a set of one or more simple instructions in LLVM's intermediate representation (IR). Fig. 7 depicts this mapping. Next, we map the IR instructions to LegUp-synthesized hardware elements. Each LLVM

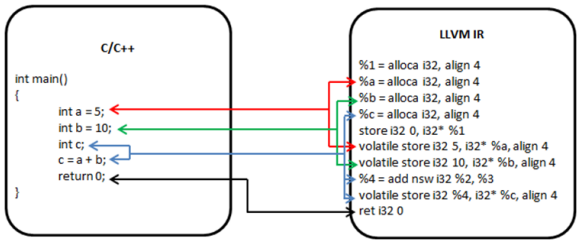


Figure 7. Mapping from C statements to LLVM intermediate representation instructions.

IR instruction is scheduled to run in one or more states of the finite state machine. Also, each IR instruction can be synthesized into several hardware units and signals. Some hardware signals, such as the memory controller signals, can be shared between multiple instructions, depending on the state.

Our goal is to have an integrated debugging system that is capable of capturing, and displaying to the user, hardware signals while running the hybrid processor/accelerator system runs on the board. Fig. 8 shows a screenshot of the LegUp debugging platform, which is "work-in-progress". Currently, the debugging platform is for simulation only; that is, we communicate with the simulation tool, ModelSim, to inspect signal values and control the simulation cycle by cycle. By examining the state of the finite state machine, we can detect the current state being executed and highlight the active C statements associated with the current state. There may more than one active C statement per state, due to the instruction-level parallelism in hardware (see Fig. 8). By clicking on a C statement, the corresponding synthesized Verilog code is highlighted. Single-stepping is supported, which runs the circuit simulation until the next C statement is reached. Note that C statements may take more than one clock cycle to complete. Developers can also *step over* a C statement to reach the next executing statement, or can *step into* a C statement to see IR-level and hardware-level details related to that statement on a cycle-by-cycle basis. Hardware signal names and current values are displayed based on the circuit's current state so that developers can track signal value changes (right panel in the figure).

The LegUp debugging platform is still under development. Supporting break-points, enabling the debugging of hybrid processor/accelerator applications and on-chip hardware debugging are all future work.

7. Conclusion

LegUp is a high-level synthesis (HLS) framework that allows software methodologies to be used for the synthesis of a hybrid system comprising an embedded processor, and one or more FPGA-based accelerators. Since the original LegUp release in March 2011, it has been downloaded over 600 times by researchers around the world (at the time of writing). As described in this paper, the current LegUp 3.0 release includes functionality to assist with hardware/software partitioning, multi-ported caches to ease memory bottlenecks, support for Pthreads and OpenMP, and improvements to the core HLS algorithms, including loop pipelining, multipumping, bitwidth optimization, and tools to select profitable compiler optimization passes to improve hardware quality. One of the few open-source frameworks of its kind, we hope the tool will be useful to the embedded systems research community as a platform to explore new design methodologies and synthesis strategies. The LegUp project website, `http://legup.eecg.toronto.edu`, includes documentation, tutorials on how to use and modify the tool, related publications, as well as links to download the source code.

8. Acknowledgements

The financial support of the Natural Sciences and Engineering Research Council of Canada (NSERC) and Altera Corporation is gratefully acknowledged.

References

[1] The OpenCL specification version: 1.0 document revision: 48, 2009.

[2] M. Aldham, J. Anderson, S. Brown, and A. Canis. Low-cost hardware profiling of run-time and energy in FPGA embedded processors. In *IEEE ASAP*, pages 61–68, 2011.

[3] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Finding effective compilation sequences. In *ACM LCTES*, pages 231–239, 2004.

[4] *Cyclone-II Data Sheet*. Altera, Corp., San Jose, CA, 2004.

[5] *DE2 Development and Education Board*. Altera, Corp., San Jose, CA, 2010.

[6] *DE4 Development Board*. Altera, Corp., San Jose, CA, 2010.

[7] *SOPC Builder User Guide*. Altera, Corp., San Jose, CA, 2010.

[8] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. Anderson, S. Brown, and T. Czajkowski. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *ACM/SIGDA FPGA*, pages 33–36, 2011.

[9] A. Canis, J. H. Anderson, and S. D. Brown. Multi-pumping for resource reduction in FPGA high-level synthesis. In *IEEE DATE*, pages 194–197, 2013.

[10] J. Choi, K. Nam, A. Canis, J. Anderson, S. Brown, and T. Czajkowski. Impact of cache architecture and interface on performance and area of FPGA-based processor/parallel-accelerator systems. In *IEEE FCCM*, pages 17–24, 2012.

[11] J. Cong and Z. Zhang. An efficient and versatile scheduling algorithm based on sdc formulation. In *ACM DAC*, volume 43, pages 433–438, 2006.

[12] J. Cong and Y. Zou. FPGA-based hardware acceleration of lithographic aerial image simulation. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2(3):1–29, 2009.

[13] P. Coussy, D. Gajski, M. Meredith, and A. Takach. An introduction to high-level synthesis. *IEEE Design Test of Computers*, 26(4):8 – 17, jul. 2009.

[14] M. Gort and J. H. Anderson. Range and bitmask analysis for hardware optimization in high-level synthesis. In *ASP DAC*, pages 773–779, 2013.

[15] 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. *Journal of Information Processing*, 17: 242–254, 2009.

[16] *Calypto Catapult*. http://calypto.com/en/products/catapult/overview, 2013.

[17] *OpenCL for Altera FPGAs*. http://www.altera.com/products/software/opencl/opencl-index.html, 2013.

[18] *C-to-Verilog*. http://www.c-to-verilog.com, 2013.

[19] *Forte Design Systems The high level design company*. http://www.forteds.com/products/cynthesizer.asp, 2013.

[20] *LLVM Compiler Infrastructure Project*. http://www.llvm.org, 2010.

[21] *Xilinx: Vivado Design Suite*. http://www.xilinx.com/products/design_tools/vivado/vivado-webpack.htm, 2013.

[22] C. Huang, Y. Che, Y. Lin, and Y. Hsu. Data path allocation based on bipartite weighted matching. In *ACM/IEEE DAC*, pages 499–504, 1990.

[23] Q. Huang, R. Lian, A. Canis, J. Choi, R. Xi, S. Brown, and J. Anderson. The effect of compiler optimizations on high-level synthesis for FPGAs. In *IEEE FCCM*, pages 89–96, 2013.

[24] H. Kuhn. The Hungarian method for the assignment problem. In *50 Years of Integer Programming 1958-2008*, pages 29–47. Springer, 2010.

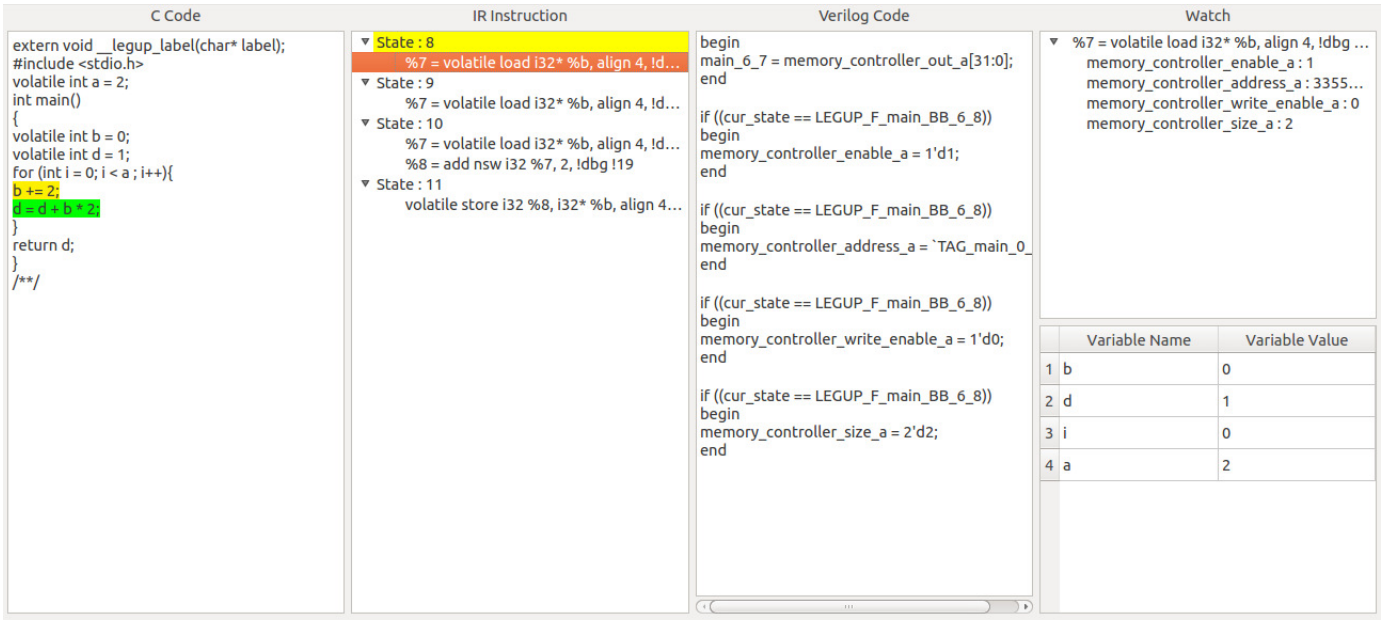


Figure 8. Screenshot of debugging platform.

[25] J. Luu, K. Redmond, W. Lo, P. Chow, L. Lilge, and J. Rose. FPGA-based monte carlo computation of light absorption for photodynamic cancer therapy. In *IEEE FCCM*, pages 157–164, 2009.

[26] *CUDA: Compute Unified Device Architecture Programming Guide*. NVIDIA CORPORATION, 2007.

[27] Z. Pan and R. Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *IEEE CGO*, pages 319–332, 2006.

[28] B. Ramakrishna Rau. Iterative modulo scheduling. *The International Journal of Parallel Processing*, 24(1):3–65, Feb 1996.

[29] R. Schreiber, S. Aditya, S. Mahlke, V. Kathail, B. R. Rau, D. Cronquist, and M. Sivaraman. PICO-NPA: High-level synthesis of nonprogrammable hardware accelerators. *Journal of VLSI signal processing systems for signal, image and video technology*, 31(2):127–142, 2002.

[30] L. Shannon and P. Chow. Using reconfigurability to achieve real-time profiling for hardware/software codesign. In *ACM FPGA*, pages 190–199, 2004.

[31] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August. Compiler optimization-space exploration. In *IEEE CGO*, pages 204–215, 2003.

[32] *Occupational Outlook Handbook 2010-2011 Edition*. United States Bureau of Labor Statistics, 2010.

[33] *The Tiger "MIPS" processor.* University of Cambridge, http://www.cl.cam.ac.uk/teaching/0910/ECAD+Arch/mips.html, 2010.

[34] *eXCite C to RTL Behavioral Synthesis 4.1(a)*. Y Explorations (XYI), San Jose, CA, 2010.