LegUp: An Open Source High-Level Synthesis Tool for FPGA-Based Processor/Accelerator Systems

ANDREW CANIS, JONGSOK CHOI, MARK ALDHAM,
VICTOR ZHANG, AHMED KAMMOONA, University of Toronto
TOMASZ CZAJKOWSKI, Altera Corporation
STEPHEN D. BROWN and JASON H. ANDERSON, University of Toronto

It is generally accepted that a custom hardware implementation of a set of computations will provide superior speed and energy-efficiency relative to a software implementation. However, the cost and difficulty of hardware design is often prohibitive, and consequently, a software approach is used for most applications. In this paper, we introduce a new high-level synthesis tool called *LegUp* that allows software techniques to be used for hardware design. LegUp accepts a standard `C` program as input and automatically compiles the program to a hybrid architecture containing an FPGA-based MIPS soft processor and custom hardware accelerators that communicate through a standard bus interface. In the hybrid processor/accelerator architecture, program segments that are unsuitable for hardware implementation can execute in software on the processor. LegUp can synthesize most of the `C` language to hardware, including fixed-sized multi-dimensional arrays, structs, global variables and pointer arithmetic. Results show that the tool produces hardware solutions of comparable quality to a commercial high-level synthesis tool. We also give results demonstrating the ability of the tool to explore the hardware/software co-design space by varying the amount of a program that runs in software vs. hardware. LegUp, along with a set of benchmark `C` programs, is open source and freely downloadable, providing a powerful platform that can be leveraged for new research on a wide range of high-level synthesis topics.

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

General Terms: Design, Algorithms

Additional Key Words and Phrases: High-level synthesis, field-programmable gate arrays, FPGAs, synthesis, performance, power, hardware/software co-design

1. INTRODUCTION

Two approaches are possible for implementing computations: software (running on a standard processor) or hardware (custom circuits). A hardware implementation can provide a significant improvement in speed and energy-efficiency versus a software implementation (e.g. [Cong and Zou 2009; Luu et al. 2009]). However, hardware design requires writing complex RTL code, which is error prone and can be notoriously difficult to debug. Software design, on the other hand, is comparatively straightforward, and mature debugging and analysis tools are freely accessible. Despite the apparent energy and performance benefits, hardware design is simply too difficult and costly for most applications, and a software approach is preferred.

This work is supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada, and Altera Corporation.
The authors are with the Dept. of Electrical and Computer Engineering, University of Toronto, Toronto, ON M5S 3G4 CANADA. T. Czajkowski is with the Altera Toronto Technology Centre, Toronto, ON M5S 1S4 CANADA. E-mail: `legup@eecg.toronto.edu`

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

In this paper, we propose *LegUp* – an open source high-level synthesis (HLS) framework we have developed that aims to provide the performance and energy benefits of hardware, while retaining the ease-of-use associated with software. LegUp automatically compiles a standard `C` program to target a hybrid FPGA-based software/hardware system-on-chip, where some program segments execute on an FPGA-based 32-bit MIPS soft processor and other program segments are automatically synthesized into FPGA circuits – *hardware accelerators* – that communicate and work in tandem with the soft processor. Since the first FPGAs appeared in the mid-1980s, access to the technology has been restricted to those with hardware design skills. However, according to labor statistics, software engineers outnumber hardware engineers by more than 10X in the U.S. [United States Bureau of Labor Statistics 2010]. An overarching goal of LegUp is to broaden the FPGA user base to include software engineers, thereby expanding the scope of FPGA applications and growing the size of the programmable hardware market – a goal we believe will keenly interest commercial FPGA vendors and the embedded systems community.

The decision to include a soft processor in the target system is based on the notion that not all `C` program code is appropriate for hardware implementation. Inherently sequential computations are well-suited for software (e.g. traversing a linked list); whereas, other computations are ideally suited for hardware (e.g. addition of integer arrays). Incorporating a processor into the target platform also offers the advantage of increased high-level language coverage – program segments that use restricted `C` language constructs can execute on the processor (e.g. calls to malloc/free). We note that most prior work on high-level hardware synthesis has focused on pure hardware implementations of `C` programs, not a hybrid software/hardware system.

LegUp is written in modular C++ to permit easy experimentation with new HLS algorithms. We leverage the state-of-the-art LLVM (low-level virtual machine) compiler framework for high-level language parsing and its standard compiler optimizations [LLVM 2010], and we implement hardware synthesis as new back-end compiler passes within LLVM. The LegUp distribution includes a set of benchmark C programs [Hara et al. 2009] that the user can compile to pure software, pure hardware, or a combined hardware/software system. For the hardware portions, LegUp produces RTL code that can be synthesized using standard commercial synthesis tools. In this paper, we present an experimental study demonstrating that LegUp produces hardware implementations of comparable quality to a commercial tool [Y Explorations (XYI) 2010]. We also give results illustrating LegUp's ability to effectively explore the design space between a pure software implementation and pure hardware implementation of a given program.

While the promise of high-level hardware synthesis has been touted for decades (consider that Synopsys introduced its Behavioral Compiler tool in 1996), the technology has yet to be embraced broadly by industry. We believe its widespread adoption has been impeded by a number of factors, including a lack of comprehensive C/C++ language support, and, in some cases, the use of non-standard languages (e.g., [Huang et al. 2008]). While a number of research groups have developed high-level hardware synthesis tools, few have gained sustained traction in the research community and the tools have been kept proprietary in many cases. The open source nature of LegUp is a key differentiator relative to prior work.

Prior high-quality open source EDA projects have had a tremendous impact in spurring new research advances. As an example, the VPR system has enabled countless studies on FPGA architecture, packing, placement, and routing [Betz and Rose 1997]. Similarly, the ABC logic synthesis system has reinvigorated low-level logic synthesis research [Mishchenko et al. 2006]. High-level hardware synthesis and application-specific processor design can likewise benefit from the availability of a robust publicly-accessible framework such as LegUp – a framework used and contributed to by researchers around the world. In fact, at the time of acceptance, the tool has been downloaded over 350 times by research groups around the world (since March 2011).

A key usage scenario for the LegUp tool is in the area of FPGA-based embedded systems design, which frequently include a soft processor [Wayne Marx 2008]. LegUp can improve computational throughput and energy-efficiency of such systems by allowing computations to be migrated from the processor to custom hardware. In addition, since LegUp can also synthesize a program (or a subset of its constituent functions) to pure hardware, it can be applied to implement the hardware accelerators in a "server style" processor/accelerator platform, where a high-end processor communicates with FPGA-based accelerators over a PCIe bus. While the server scenario is certainly possible, it is the embedded systems usage model that is explored more heavily in this paper.

A preliminary version of a portion of this work appears in [Canis et al. 2011]. In this extended journal version, we elaborate on all aspects of the proposed framework, including background on the intermediate representation (IR) within the LLVM compiler, and how programs represented in the IR are synthesized to hardware circuits. We describe the processor/accelerator interconnection approach in further detail, as well as provide additional information on the benchmark suite and debugging capabilities. Circuit-by-circuit experimental results for speed, area and power are also included (whereas, only average data was included in the 4-page conference version). We also describe how LegUp can be modified to support different FPGA architectures, implement a new scheduling algorithm, and support parallel accelerators.

The remainder of this paper is organized as follows: Section 2 presents related work. Section 3 introduces the target hardware architecture and outlines the high-level design flow. The details of the high-level synthesis tool and software/hardware partitioning are described in Section 4. An experimental evaluation appears in Section 5. Section 6 presents three cases studies that serve to demonstrate the extensibility of the LegUp tool: 1) to target an alternate FPGA device, 2) to evaluate a different scheduling algorithm, and 3) to support concurrently running accelerators. Conclusions and suggestions for future work are given in Section 7.

2. RELATED WORK

2.1. High-Level Synthesis

Automatic compilation of a high-level language program to silicon has been a decades-long quest in the EDA field, with early seminal work done in the 1980s. We highlight several recent efforts, with emphasis on tools that target FPGAs.

Several HLS tools have been developed for targeting specific applications. GAUT is a high-level synthesis tool that is designed for DSP applications [Coussy et al. 2010]. GAUT synthesizes a `C` program into an architecture with a processing unit, a memory unit, and a communication unit, and requires that the user supply specific constraints, such as the pipeline initiation interval.

ROCCC is an open source high level synthesis tool that can create hardware accelerators from `C` [Villarreal et al. 2010]. ROCCC is designed to accelerate critical kernels that perform repeated computation on streams of data, for instance DSP applications such as FIR filters. ROCCC does not support several commonly-used aspects of the `C` language, such as generic pointers, shifting by a variable amount, non-for loops, and the ternary operator. ROCCC has a bottom-up development process that involves partitioning one's application into *modules* and *systems*. Modules are `C` functions that are converted into computational datapaths with no FSM, with loops fully unrolled. These modules cannot access memory but have data pushed to them and output scalar values. Systems are `C` functions that instantiate modules to repeat computation on a stream of data or a window of memory, and usually consist of a loop nest with special function parameters for streams. ROCCC supports advanced optimizations such as systolic array generation, temporal common subexpression elimination, and it can generate Xilinx PCore modules to be used with a Xilinx MicroBlaze proces-

Table I. Release status of recent non-commercial HLS tools.

Open source	Binary only	No source or binary
Trident	xPilot	WarpProcessor
ROCCC	GAUT	LiquidMetal
		CHiMPS

sor. However, ROCCC's strict subset of `C` is insufficient for compiling any of the CHStone benchmarks used in this study and described in Section 4.5. Broadly speaking, ROCCC works and excels for a specific class of applications (streaming-oriented applications), but it is not a general C-to-hardware compiler. By supporting the CHStone benchmarks, LegUp provides researchers with the opportunity to compile larger C programs than is possible with ROCCC.

General (application-agnostic) tools have also been proposed in recent years. CHiMPS is a tool developed by Xilinx and the University of Washington that synthesizes programs into a *many cache* architecture, taking advantage of the abundant small block RAMs available throughout the FPGA fabric [Putnam et al. 2008]. LiquidMetal is a compiler being developed at IBM Research comprising a HLS compiler and a new (non-standard) language, *LIME*, that incorporates hardware-specific constructs, such as bitwidth specification on integers [Huang et al. 2008]. xPilot is a tool that was developed at UCLA [Cong et al. 2006] and used successfully for a number of HLS studies (e.g., [Chen and Cong 2004]). Trident is a tool developed at Los Alamos National Labs, with a focus on supporting floating point operations [Tripp et al. 2007]. xPilot and Trident have not been under active development for several years and are no longer maintained.

Among prior academic work, the *Warp Processor* proposed by Vahid, Stitt and Lysecky bears the most similarity to our framework [Vahid et al. 2008]. In a Warp Processor, software running on a processor is profiled during its execution. The profiling results guide the selection of program segments to be synthesized to hardware. Such segments are disassembled from the software binary to a higher-level representation, which is then synthesized to hardware [Stitt and Vahid 2007]. The software binary running on the processor is altered automatically to leverage the generated hardware. We take a somewhat similar approach, with the key differences being that we compile hardware from the high-level language source code (not from a disassembled binary) and our tool is open source.

With regard to commercial tools, there has been considerable activity in recent years, both in start-ups and major EDA vendors. Current offerings include AutoPilot from AutoESL [AutoESL] (a commercial version of xPilot, recently acquired by Xilinx, Inc.), Catapult `C` from Mentor Graphics [Mentor Graphics 2010], C2R from CebaTech [CebaTech 2010], eXCite from Y Explorations [Y Explorations (XYI) 2010], CoDeveloper from Impulse Accelerated Technologies [Impulse 2010], Cynthesizer from Forte [Forte 2010], and C-to-Silicon from Cadence [Cadence 2010]. On our experience, attaining a binary executable for evaluation has not been possible for most tools.

Also on the commercial front is Altera's C2H tool [Altera, Corp. 2009]. C2H allows a user to partition a `C` program's functions into a hardware set and a software set, where the software-designated functions execute on a Nios II soft processor, and the hardware-designated functions are synthesized into custom hardware accelerators that connect to the Nios II through an Avalon interface (Altera's on-chip interconnect standard). The C2H target system architecture closely resembles that targeted by our tool.

Table I shows the release status of each non-commercial tool surveyed above, indicating whether each is: 1) open source, 2) binary only (i.e., only the binary is publicly available), or 3) no source or binary available. Tools in category #2 cannot be modified by the research community to explore new HLS algorithms or new processor/accelerator design styles. Results produced by tools in category #3 cannot be independently replicated. In the open

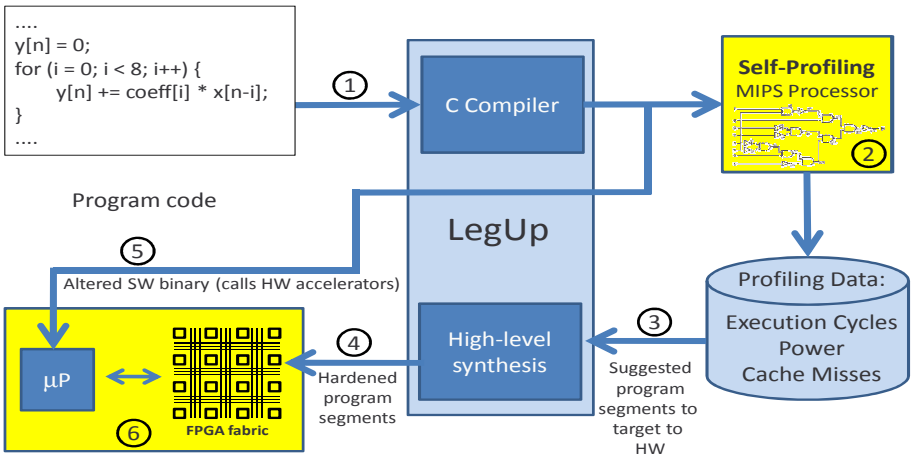

Fig. 1. Design flow with LegUp.

source category, the Trident tool was based on an early version of LLVM, however, it is has not been actively maintained for several years, and it targeted pure hardware and not a hybrid hardware/processor architecture. ROCCC is actively being worked on, however, it targets a feed-forward pipeline hardware architecture model. To our knowledge, there is currently no open source HLS tool that compiles a standard `C` program to a hybrid processor/accelerator system architecture, where the synthesized hardware follows a general datapath/state machine model. By supporting nearly all of the commonly-used aspects of the C language, as evidenced by the CHStone benchmark programs [Hara et al. 2009], LegUp provides researchers with the infrastructure needed to compile larger and more general C programs than those supported by ROCCC. Section 6 describes case studies that demonstrate the tools extensibility.

2.2. Application-Specific Instruction Processors (ASIPs)

The concept of an application-specific instruction set processor (ASIP) (e.g. [Pothineni et al. 2010], [Pozzi et al. 2006], [Henkel 2003], [Sun et al. 2004]) is also related to the hybrid processor/accelerator platform targeted by LegUp. An ASIP combines a processor with custom hardware to improve the speed and energy efficiency of an application. In an ASIP, a cluster of a program's instructions is selected, and the cluster is replaced by a (new) custom instruction that *calls* a custom hardware unit implementing the functionality of the cluster. There are two main differences between typical ASIPs and the LegUp platform. First accelerators in ASIPs are closely coupled to the processor. While the coupling allows the accelerators to access the processor's registers, it requires the processor to stall while an accelerator performs computation. Thus, performance is limited in comparison to LegUp, where the loosely-coupled processor/accelerator architecture permits multiple accelerators and the processor to execute concurrently. Second, the LegUp platform is a full HLS framework capable of synthesizing an entire program to hardware and is not limited to synthesizing clusters of instructions.

3. LEGUP OVERVIEW

In this section, we provide a high-level overview of the LegUp design flow and its target architecture. Algorithmic and implementation details follow in Section 4.

3.1. Design Flow

The LegUp design flow comprises first compiling and running a program on a standard processor, profiling its execution, selecting program segments to target to hardware, and then re-compiling the program to a hybrid hardware/software system. Figure 1 illustrates the detailed flow. Referring to the labels in the figure, at step ①, the user compiles a standard `C` program to a binary executable using the LLVM compiler. At ②, the executable is run on an FPGA-based MIPS processor. We selected the Tiger MIPS processor from the University of Cambridge [University of Cambridge 2010], based on its support for the full MIPS instruction set, established tool flow, and well-documented modular Verilog.

The MIPS processor has been augmented with extra circuitry to profile its own execution. Using its profiling ability, the processor is able to identify sections of program code that would benefit from hardware implementation, improving program throughput and power. Specifically, the profiling results drive the selection of program code segments to be re-targeted to custom hardware from the `C` source. Profiling a program's execution in the processor itself provides the highest possible accuracy, as the executing code does not need to be altered to be profiled and can run at full speed. Moreover, with hardware profiling, system-level characteristics that affect performance are properly accounted for, such as off-chip memory access times. In this paper, we profile program run-time at the function level. In the first release of our tool the user must manually examine the profiling results and place the names of the functions to be accelerated in a `Tcl` file that is read by LegUp.

Having chosen program segments to target to custom hardware, at step ③ LegUp is invoked to compile these segments to synthesizeable Verilog RTL. Presently, LegUp HLS operates at the function level: entire functions are synthesized to hardware from the C source. Moreover, if a hardware function calls other functions, such called functions are also synthesized to hardware. In other words, we do not allow a hardware-accelerated function to call a software function. The RTL produced by LegUp is synthesized to an FPGA implementation using standard commercial tools at step ④. As illustrated in the figure, LegUp's hardware synthesis and software compilation are part of the same LLVM-based compiler framework.

In step ⑤, the `C` source is modified such that the functions implemented as hardware accelerators are *replaced* by wrapper functions that *call* the accelerators (instead of doing computations in software). This new modified source is compiled to a MIPS binary executable. Finally, in step ⑥ the hybrid processor/accelerator system executes on the FPGA.

3.2. Target System Architecture

Figure 2 elaborates on the target system architecture. The processor connects to one or more custom hardware accelerators through a standard on-chip interface. As our initial hardware platform is the Altera DE2 Development and Education board (containing a 90nm Cyclone II FPGA) [DE2 2010], we use the Altera Avalon interface for processor/accelerator communication [Altera, Corp. 2010]. Synthesizable RTL code for the Avalon interface is generated automatically using Altera's SOPC builder tool. The Avalon interface comprises point-to-point connections between communicating modules – it is not a shared bus. The Cyclone II/DE2 was chosen because of its widespread availability.

As shown in Figure 2, a shared memory architecture is used, with the processor and accelerators sharing an on-FPGA data cache and off-chip main memory (8 MB of SDRAM). The on-chip cache memory is implemented using block RAMs within the FPGA fabric (M4K blocks on Cyclone II). Access to memory is handled by a memory controller. Such an architecture allows processor/accelerator communication across the Avalon interface or through memory. The shared single cache obviates the need to implement cache coherency or automatic cache line invalidation. Although not shown in the figure, the MIPS soft processor also has an instruction cache.

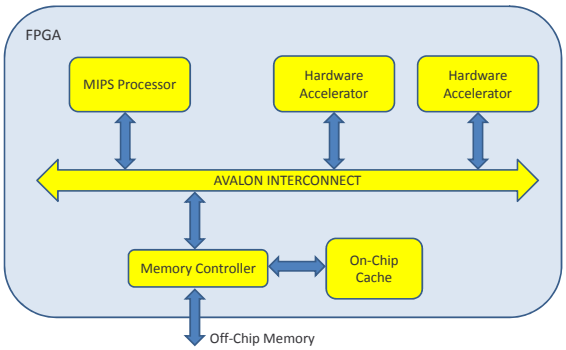

Fig. 2. Target system architecture.

The architecture depicted in Figure 2 represents the target system most natural for an initial release of the tool. We expect the shared memory to become a bottleneck if many processors and accelerators are included in the system. The architecture of processor/accelerator systems is an important direction for future research – research enabled by a framework such as LegUp – with key questions being the investigation of the best on-chip connectivity and memory architecture. Moreover, in our initial release, the processor and accelerators share a single clock signal. Multi-clock domain processor/accelerator systems-on-chip is an important avenue to explore.

4. DESIGN AND IMPLEMENTATION

4.1. High-Level Hardware Synthesis

High-level synthesis has traditionally been divided into three steps [Coussy et al. 2009]: allocation, scheduling and binding. Allocation determines the amount of hardware available for use (e.g., the number of adder functional units), and also manages other hardware constraints (e.g., speed, area, and power). Scheduling assigns each operation in the program being synthesized to a particular clock cycle (state) and generates a finite state machine. Binding assigns a program's operations to specific hardware units. The decisions made by binding may imply sharing functional units between operations, and sharing registers/memories between variables. We now describe our initial implementation choices for the HLS steps, beginning with a discussion of the compiler infrastructure.

4.1.1. Low-Level Virtual Machine (LLVM). LegUp leverages the low-level virtual machine (LLVM) compiler framework – the same framework used by Apple for iPhone/iPad application development. At the core of LLVM is an intermediate representation (IR), which is essentially machine-independent assembly language. C code is translated into LLVM's IR then analyzed and modified by a series of compiler optimization passes. Current results show that LLVM produces code of comparable quality to `gcc` for x86-based processor architectures.

Consider an 8-tap finite impulse response (FIR) filter whose output, `y[n]`, is a weighted sum of the current input sample, `x[n]` and seven previous input samples. The `C` code for calculating the FIR response is given in Figure 3. The unoptimized LLVM IR corresponding to this `C` code is given in Figure 4. We highlight a few key elements of the IR here. The LLVM IR is in single static assignment (SSA) form, which prohibits variable re-use, guaranteeing a 1-to-1 correspondence between an instruction and its destination register. Register names in the IR are prefixed by `%`. Types are explicit in the IR. For example, `i32` specifies a 32-bit integer type and `i32*` specifies a pointer to a 32-bit integer.

```
y[n] = 0;
for(i = 0; i < 8; i++) {
    y[n] += coeff[i] * x[n - i];
}
```

Fig. 3. C code for FIR filter.

In the example IR for the FIR filter in Figure 4, line 1 marks the beginning of a *basic block* called `entry`. A basic block is a contiguous set of instructions with a single entry (at its beginning) and exit point (at its end). Lines 2 and 3 initialize `y[n]` to 0. Line 4 is an unconditional branch to a basic block called `bb1` that begins on line 5. `phi` instructions are needed to handle control flow-dependent variables in SSA form. For example, the `phi` instruction on line 6 assigns loop index register `%i` to 0 if the previous basic block was *entry*; otherwise, `%i` is assigned to register `%i.new`, which contains the incremented `%i` from the previous loop iteration. Line 7 initializes a pointer to the coefficient array. Lines 8 and 9 initialize a pointer to the sample array `x`. Lines 10-12 load the sum `y[n]`, sample and coefficient into registers. Lines 13 and 14 perform the multiply-accumulate. The result is stored in line 15. Line 16 increments the loop index `%i`. Lines 17 and 18 compare `%i` with loop limit (8) and branch accordingly.

Observe that LLVM instructions are simple enough to directly correspond to hardware operations (e.g., a load from memory, or an arithmetic computation). Our HLS tool operates directly with the LLVM IR, scheduling the instructions into specific clock cycles (described below).

Scheduling operations in hardware requires knowing data dependencies between operations. Fortunately, the SSA form of the LLVM IR makes this easy. For example, the multiply instruction (`mul`) on line 13 of Figure 4 depends on the results of two load instructions on lines 11 and 12. Memory data dependencies are more problematic to discern; however, LLVM includes alias analysis – a compiler technique for determining which memory locations a pointer can reference. In Figure 4, the `store` on line 15 has a write-after-read dependency with the load on line 10, but has no memory dependencies with the loads on lines 12 and 13. Alias analysis can determine that these instructions are independent and can therefore be performed in parallel.

Transformations and optimizations in the LLVM framework are structured as a series of compiler passes. Passes include optimizations such as dead code elimination, analysis passes such as alias analysis, and back-end passes that produce assembly for a particular target machine (e.g. MIPS or ARM). The infrastructure is flexible, allowing passes to be reordered, substituted with alternatives, and disabled. LegUp HLS algorithms have been implemented as LLVM passes that fit into the existing framework. Implementing the HLS steps as distinct passes also allows easy experimentation with different HLS algorithms. For example, one could modify LegUp to "plug in" a new scheduling algorithm and study its impact on quality of results.

4.1.2. Device Characterization. For a given FPGA family, LegUp includes scripts to pre-characterize the hardware operation corresponding to each LLVM instruction for all supported bitwidths (typically, 8, 16, 32, 64). The scripts synthesize each operation in isolation for the target FPGA family to determine the propagation delay, required number of logic elements, registers, multiplier blocks, and power consumption. This characterization data allows LegUp to make early predictions of circuit speed and area for the hardware accelerators and also to aid scheduling and binding.

4.1.3. Allocation. The purpose of allocation is to determine the amount of hardware that may be used to implement the circuit. LegUp reads allocation information from a configuration `Tcl` file, which specifies the target FPGA device and the resource limits for the device,

```
1: entry:
2:   %y.addr = getelementptr i32* %y, i32 %n
3:   store i32 0, i32* %y.addr
4:   br label %bb1
5: bb1:
6:   %i = phi i32 [ 0, %entry ], [ %i.new, %bb1 ]
7:   %coeff.addr = getelementptr [8 x i32]* %coeff,
                  i32 0, i32 %i
8:   %x.ind = sub i32 %n, %i
9:   %x.addr = getelementptr i32* %x, i32 %x.ind
10:  %0 = load i32* %y.addr
11:  %1 = load i32* %coeff.addr
12:  %2 = load i32* %x.addr
13:  %3 = mul i32 %1, %2
14:  %4 = add i32 %0, %3
15:  store i32 %4, i32* %y.addr
16:  %i.new = add i32 %i, 1
17:  %exitcond = icmp eq i32 %i.new, 8
18:  br i1 %exitcond, label %return, label %bb1
19:return:
```

Fig. 4. LLVM IR for FIR filter.

e.g. the number of available multiplier blocks. In general, LegUp HLS operates as though an unlimited amount of resources are available in the target FPGA. The reason for this is that resource sharing (i.e. using a single hardware unit to implement multiple operations within the program being synthesized) requires adding multiplexers to the input ports of a shared hardware unit, and multiplexers are costly to implement in FPGAs. For example, a 32-bit adder can be implemented using 32 4-input LUTs (and associated carry logic), and 32 2-to-1 multiplexers also require 32 4-input LUTs – the same number of LUTs as the adder itself! Thus, for the allocation step, LegUp does the following:

— Multiply: Hard multiplier blocks in the FPGA fabric are used. Sharing multipliers is only done when the benchmark being synthesized requires more multipliers than that available in the FPGA.
— Divide/Modulus: These operations are implemented with LUTs, and consume significant area. Therefore, we set the number of divide/remainder units to be the maximum number used in any cycle of the schedule. Multiplexers are added to the input ports of the unit(s) to facilitate the resource sharing (described below in the binding section).

4.1.4. Scheduling. Scheduling is the task of assigning operations to clock cycles and building a finite state machine (FSM). A control flow graph (CFG) of a program is a directed graph where basic blocks are represented by vertices and branches are represented by edges. For example, given two basic blocks, b_1 and b_2, b_1 has an edge to b_2 in the CFG if b_1 can branch to b_2. We can think of a CFG as a *coarse* representation of the FSM needed to control the hardware being synthesized – the nodes and edges are analogous to those of a state diagram. What is not represented in this coarse FSM are data dependencies between operations within a basic block and the latencies of operations (e.g., a memory access may take more than a single cycle).

Having constructed the coarse FSM from the CFG, LegUp then schedules each basic block individually, which amounts to splitting each node in the CFG into multiple nodes, each corresponding to one FSM state (clock cycle). The initial release of LegUp uses as-soon-as-possible (ASAP) scheduling [Gajski and et. al. Editors 1992], which assigns an instruction to the first state after all of its dependencies have been computed. Traversing basic blocks, and visiting the instructions within each basic block in order, the operands for each instruction

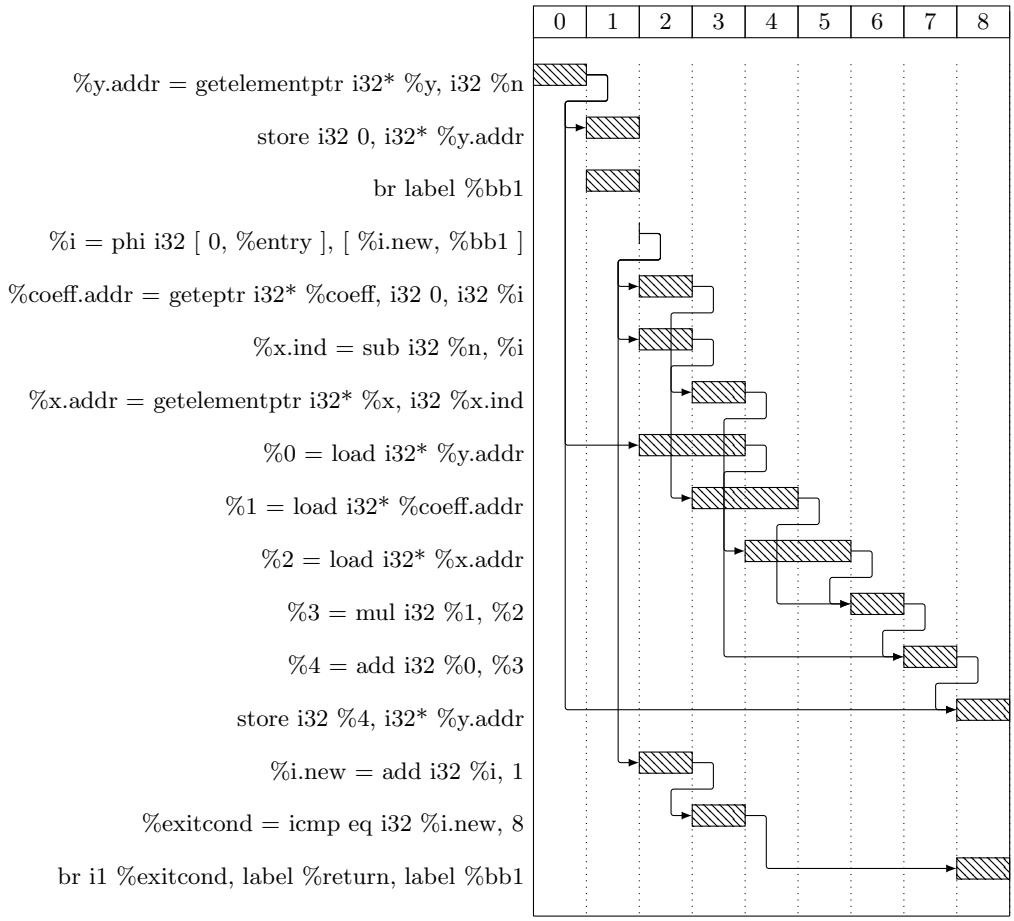

Fig. 5. Scheduled FIR filter IR with data dependencies.

are either: 1) from this basic block and therefore guaranteed to have already been assigned a state, or 2) from outside this basic block, in which case we can safely assume they will be available before control reaches this basic block. Note that our scheduler properly handles instructions with multi-cycle latencies, such as pipelined divides or memory accesses.

In some cases, we can schedule an instruction into the *same* state as one of its operands. This is called *operation chaining*. We perform chaining in cases where the estimated delay of the chained operations (from allocation) does not exceed the estimated clock period for the design. Chaining can reduce hardware latency (# of cycles for execution) and save registers without impacting the final clock period.

Fig. 5 is a Gantt chart showing the ASAP schedule of the FIR filter instructions shown in Fig. 4. The chart shows the same LLVM instructions, now organized into nine states. Data dependencies between operations are shown; in this case we do not allow operation chaining (for clarity). Load instructions have a two cycle latency, allowing us to pipeline our memory controller for higher speed performance. Once a load has been issued, a new load can be issued on the next cycle. Because our memory controller is single ported, only one load can be performed every cycle.

4.1.5. Binding. Binding comprises two tasks: assigning operators from the program being synthesized to specific hardware units (operation assignment), and assigning program variables to registers (register allocation). When multiple operators are assigned to the same hardware unit, or when multiple variables are bound to the same register, multiplexers are required to facilitate the sharing. We make two FPGA-specific observations in our approach to binding. First, multiplexers are relatively expensive to implement in FPGAs using LUTs. Consequently, there is little advantage to sharing all but the largest functional units, namely, multipliers and dividers. Likewise, the FPGA fabric is *register rich* – each logic element in the fabric has a LUT and a register. Therefore, sharing registers is rarely justified.

We have three goals when binding operations to shared functional units. First, we would like to balance the sizes of the multiplexers across functional units to keep circuit performance high. Multiplexers with more inputs have higher delay, so it is desirable to avoid having a functional unit with a disproportionately large multiplexer on its input. Second, we want to recognize cases where we have shared inputs between operations, letting us save a multiplexer if the operations are assigned to the same functional unit. Lastly, during binding if we can assign two operations that have non-overlapping livetime intervals to the same functional unit, we can use a *single* output register for both operations. In this case we save a register, without needing a multiplexer. We use the LLVM live variable analysis pass to check for the livetime intervals.

To account for these goals we use the following cost function to measure the benefit of assigning operation *op* to function unit *fu*:

$$\begin{aligned} Cost(op, fu) \;=\; & \phi \cdot existingMuxInputs(fu) + \\ & \beta \cdot newMuxInputs(op, fu) - \\ & \theta \cdot outputRegisterSharable(op, fu) \end{aligned} \tag{1}$$

where $\phi = 0.1$, $\beta = 1$, and $\theta = 0.5$ to give priority to saving new multiplexer inputs, then output registers, and finally balancing the multiplexers. Notice that sharing the output register reduces the cost, while the other factors increase it.

The initial release of LegUp uses a weighted bipartite matching heuristic to solve the binding problem [Huang et al.]. The binding problem is represented using a bipartite graph with two vertex sets. The first vertex set corresponds to the operations being bound (i.e. LLVM instructions). The second vertex set corresponds to the available functional units. A weighted edge is introduced from a vertex in the first set to a vertex in the second set if the corresponding operation is a candidate to be bound to the corresponding functional unit. We set the cost (edge weight) of assigning an operation to a functional unit using (1). Weighted bipartite matching can be solved optimally in polynomial time using the well-known Hungarian method [Kuhn 2010]. We formulate and solve the matching problem one clock cycle at a time until the operations in all clock cycles (states) have been bound.

4.2. Local Accelerator Memories

The system architecture shown in Figure 2 includes a shared memory between the processor and accelerators, comprising on-FPGA cache and off-FPGA SDRAM. Accesses to the off-chip SDRAM are detrimental to performance, as each access takes multiple clock cycles to complete, and contention may arise in the case of concurrent accesses. To help mitigate this, constants and local variables used within hardware accelerators (which are not shared with the processor) are stored in block RAMs in the accelerators themselves. We create local memories for each variable/constant array used by an accelerator. An advantage of using multiple memories instead of a single large memory is enhanced parallelism.

Each local memory is assigned a 9-bit tag using the top 9 bits of the 32-bit address space. The tag is used to steer a memory access to the correct local accelerator memory, or alternately, to the shared memory, that is, to the memory controller shown in Figure 2.

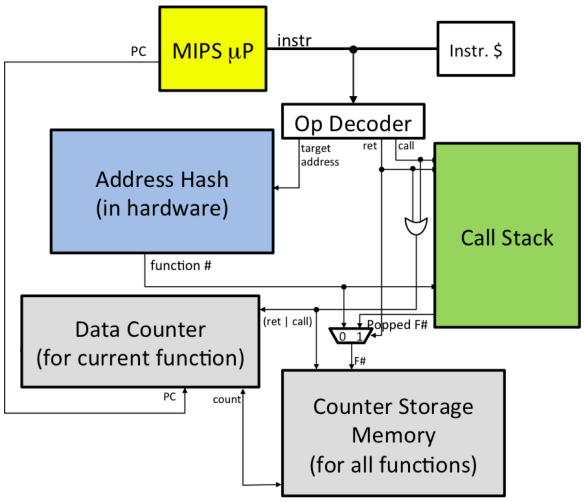

Fig. 6. Profiler hardware architecture.

LegUp automatically generates the multiplexing logic to interpret the tags and steer memory requests. Tag `000000000` is reserved for the `NULL` pointer, and tag `000000001` indicates that the memory access should be steered to the shared memory. The remaining 510 different tags can be used to differentiate between up to 510 local accelerator memories. Using 9 bits for the tag implies that 23 bits are available for encoding the address. The decision to use 9-bit tags in the initial release of LegUp was taken because the Altera DE2 board contains an 8 MB SDRAM which is fully addressable using 23 bits. It is straightforward to change LegUp to use a different tag width if desired.

4.3. Hardware Profiling

As shown in Figure 1, a hardware profiler is used to decide which functions should be implemented as hardware accelerators. The profiler utilized in LegUp is a non-intrusive, real-time profiler that performs its analyses at the function level. As a program executes on the MIPS processor, the profiler monitors the program counter and instruction op-codes to track the number of cycles spent in each function and its descendants.

At a high-level, our profiler works by associating both an index and a counter with each function in a program. The index for a function is computed using a hash of the memory address of the function's first instruction. The hash can be calculated in hardware using simple logical and arithmetic operations. The counter tracks the total number of execution cycles spent in the function and optionally, execution cycles spent in the function's descendants. The number of functions being tracked by the profiler is configurable, as are the widths of the cycle counters. Most importantly, the profiler allows different programs to be profiled without requiring any re-synthesis.

Fig. 6 provides an overview of the profiler architecture. An operation decoder module (labeled *Op Decoder*) monitors instructions issued by the processor, looking for function calls and returns. When a call is detected, the destination address is passed to the *Address Hash* module which computes a unique index for the called function. The function index is pushed onto a stack in the *Call Stack* module (the stack is implemented in FPGA block RAM). A *Data Counter* module accumulates profiling data for the current function being executed, while the *Counter Storage* contains profiling data for all functions in the program. Pushing and popping function indices onto/from the stack on function calls and returns allows profiling data to be accrued to the appropriate functions. The profiler represents a

```
int add (int * a, int * b, int N)
{
  int sum=0;
  for (int i=0; i<N; i++)
  {
    sum += a[i]+b[i];
  }
  return sum;
}
```

Fig. 7. C function targeted for hardware.

6.7% overhead on the MIPS processor area when configured to track up to 32 functions using 32-bit counters. Complete details on the profiler, including how it can be extended to profile energy consumption, are omitted for lack of space, but can be found in [Aldham et al. 2011].

4.4. Processor/Accelerator Communication

Recall the target architecture shown in Figure 2 comprising a MIPS processor that communicates with hardware accelerators. When a function is selected to be implemented in hardware, its `C` implementation is automatically replaced with a wrapper function by the LegUp framework. The wrapper function passes the function arguments to the corresponding hardware accelerator, asserts a start signal to the accelerator, waits until the accelerator has completed execution, and then receives the returned data over the Avalon interconnection fabric.

The MIPS processor can do one of two things while waiting for the accelerator to complete its work: 1) it can continue to perform computations and periodically poll a memory-mapped register whose value is set to `1` when the accelerator is done, or, 2) it can stall until a done signal is asserted by the accelerator. The advantage of polling is that the processor can execute other computations concurrent with the accelerator doing its work, akin to a threaded computing environment. The advantage of stalling is energy consumption – the processor is in a low-power state while the accelerator operates. In our initial LegUp release, both modes are functional; however, we use only mode #2 (stalling) for the results in this paper.

To illustrate the wrapper concept, consider the `C` function shown in Figure 7. The function accepts two N-element vectors as input and computes the sum of the vectors' pairwise elements. If function is to be implemented in hardware, it would be replaced with the wrapper function shown in Figure 8. The defined memory addresses correspond to the assigned memory space of the hardware accelerator. Each accelerator contains logic to communicate with the processor according to the signals and addresses asserted through the Avalon interconnect. Writes to the specified memory addresses translate into data communicated across the Avalon interface to the accelerator. The write to the `STATUS` address starts the accelerator. At this point, the accelerator asserts an input signal to the processor causing it to stall; the accelerator de-asserts this signal when its work is complete. A read from the `DATA` address retrieves the vector addition result from the accelerator.

4.5. Language Support and Benchmarks

LegUp supports a large subset of ANSI `C` for synthesis to hardware including: function calls, assignments, loops, integer logical, bitwise and arithmetic operations. Program segments that use unsupported language features are required to remain in software and execute on the MIPS processor. Table II lists `C` language constructs that are frequently problematic for hardware synthesis, and specifies which constructs are supported/unsupported by LegUp.

```
#define STATUS   (volatile int *)0xf00000000
#define DATA  (volatile int *)0xf00000004
#define ARG1 (volatile int *)0xf00000008
#define ARG2 (volatile int *)0xf0000000C
#define ARG3 (volatile int *)0xf00000010

int add (int * a, int * b, int N)
{
  // pass arguments to accelerator
  *ARG1 = a;
  *ARG2 = b;
  *ARG3 = N;
  // give start signal
  *STATUS = 1;
  // wake up and get return data
  return *DATA;
}
```

Fig. 8. Wrapper for function in Figure 7.

Table II. Language support.

Supported	**Unsupported**
Functions	Dynamic Memory
Arrays, Structs	Floating Point
Global Variables	Recursion
Pointer Arithmetic	

Table III. Benchmark programs included with LegUp.

Category	**Benchmarks**	**Lines of C**
Arithmetic	64-bit dbl precision add, mult, div, sin	376–755
Encryption	AES, Blowfish, SHA	716–1406
Processor	MIPS processor	232
Media	JPEG decoder, Motion, GSM, ADPCM	393–1692
General	Dhrystone	491

Unlike many HLS tools, synthesis of fixed-size multi-dimensional arrays, structs, global variables, and pointer arithmetic are supported by LegUp. Regarding structs, LegUp supports structs with arrays, arrays of structs, and structs containing pointers. LegUp stores structs in memory using the ANSI C alignment standards. Functions that return a struct, dynamic memory allocation, recursion and floating point arithmetic are unsupported in the initial release of the tool.

With the LegUp distribution, we include 13 benchmark C programs, summarized in Table III. Included are all 12 programs in the CHStone high-level synthesis benchmark suite [Hara et al. 2009], as well as Dhrystone – a standard integer benchmark. The programs represent a diverse set of computations falling into several categories: arithmetic, encryption, media, processing and general. They range in size from 232-1692 lines of C code. The arithmetic benchmarks implement 64-bit double-precision floating-point operations in software using integer types. Notice that the CHStone suite contains a benchmark which is a software model of a MIPS processor (which we can then run on a MIPS processor).

A key characteristic of the benchmarks is that inputs and expected outputs are included in the programs themselves. The presence of the inputs and golden outputs for each program gives us assurance regarding the correctness of our synthesis results. Each benchmark program performs computations whose results are then checked against golden values. This is analogous to built-in self test in design-for-test methodology. No inputs (e.g. from the keyboard or a file) are required to run the programs.

4.6. Debugging

The initial release of LegUp includes a basic debugging capability which consists of automatically adding print statements into the LLVM IR to dump variable values at the end of each basic block's execution. When the IR is synthesized to hardware, the Verilog can be simulated using ModelSim producing a log of variable value changes that can be directly compared with an analogous log from a strictly software execution of a benchmark. We found even this limited capability to be quite useful, as it allows one to pinpoint the first LLVM instruction where computed values differ in hardware vs. software, aiding problem diagnosis and debugging.

5. EXPERIMENTS

The goals of our experimental study are three-fold: 1) to demonstrate that the quality of results (speed, area, power) produced by LegUp HLS is comparable to that produced by a commercial HLS tool, eXCite [Y Explorations (XYI) 2010], 2) to demonstrate LegUp's ability to effectively explore the hardware/software co-design space, and 3) to compare the quality of hardware vs. software implementations of the benchmark programs. We chose eXCite because it was the only commercial tool we had access to that could compile the benchmark programs. With the above goals in mind, we map each benchmark program using 5 different flows, representing implementations with successively increasing amounts of computation happening in hardware vs. software. The flows are as follows (labels appear in parentheses):

(1) A software-only implementation running on the MIPS soft processor (*MIPS-SW*).
(2) A hybrid software/hardware implementation where the *second most*[1] compute-intensive function (and its descendants) in the benchmark is implemented as a hardware accelerator, with the balance of the benchmark running in software on the MIPS processor (*LegUp-Hybrid2*).
(3) A hybrid software/hardware implementation where the *most* compute-intensive function (and its descendants) is implemented as a hardware accelerator, with the balance in software (*LegUp-Hybrid1*).
(4) A pure hardware implementation produced by LegUp (*LegUp-HW*).
(5) A pure hardware implementation produced by eXCite (*eXCite-HW*)[2].

The two hybrid flows correspond to a system that includes the MIPS processor and a single accelerator, where the accelerator implements a `C` function and all of its descendant functions.

For the back-end of the flow, we use Quartus II ver. 9.1 SP2 to target the Cyclone II FPGA. Quartus II was executed in timing-driven mode with all physical synthesis optimizations turned on[3]. The correctness of the LegUp implementations was verified using post-routed ModelSim simulations and also in hardware using the Altera DE2 board.

[1]Not considering the `main()` function.
[2]The eXCite implementations were produced by running the tool with the default options.
[3]The eXCite implementation for the *jpeg* benchmark was run without physical synthesis optimizations turned on in Quartus II, as with such optimizations, the benchmark could not fit into the largest Cyclone II device.

Table IV. Speed performance results (frequencies in MHz, times in μS)

	MIPS-SW			LegUp-Hybrid2			LegUp-Hybrid1			LegUp-HW			eXCite-HW		
Benchmark	**Cycles**	**Freq.**	**Time**	**Cycles**	**Freq.**	**Time**	**Cycles**	**Freq.**	**Time**	**Cycles**	**Freq.**	**Time**	**Cycles**	**Freq.**	**Time**
adpcm	193607	74.26	2607	159883	61.61	2595	96948	57.19	1695	36795	45.79	804	21992	28.88	761
aes	73777	74.26	993	55014	54.97	1001	26878	49.52	543	14022	60.72	231	55679	50.96	1093
blowfish	954563	74.26	12854	680343	63.21	10763	319931	63.7	5022	209866	65.41	3208	209614	35.86	5845
dfadd	16496	74.26	222	14672	75.01	196	5649	77.41	73	2330	124.05	19	370	24.54	15
dfdiv	71507	74.26	963	15973	77.92	205	4538	65.92	69	2144	74.72	29	2029	43.95	46
dfmul	6796	74.26	92	10784	75.58	143	2471	79.14	31	347	85.62	4	223	49.17	5
dfsin	2993369	74.26	40309	293031	65.66	4463	80678	68.23	1182	67466	62.64	1077	49709	40.06	1241
gsm	39108	74.26	527	29500	61.46	480	18505	61.14	303	6656	58.93	113	5739	41.82	137
jpeg	29802639	74.26	401328	16072954	51.2	313925	15978127	46.65	342511	5861516	47.09	124475	3248488	22.66	143358
mips	43384	74.26	584	6463	75.51	86	6463	75.51	86	6443	90.09	72	4344	76.25	57
motion	36753	74.26	495	34859	73.34	475	17017	79.67	214	8578	91.79	93	2268	42.87	53
sha	1209523	74.26	16288	358405	77.40	4631	265221	75.76	3508	247738	86.93	2850	238009	62.48	3809
dhrystone	28855	74.26	389	25599	77.64	330	25509	76.99	331	10202	85.38	119	-	-	-
Geomean:	**173332.0**	**74.26**	**2334.1**	**86258.3**	**67.10**	**1285.9**	**42700.5**	**65.65**	**650.3**	**20853.8**	**71.56**	**291.7**	**14594.4**	**40.87**	**357.1**
Ratio:	**1**	**1**	**1**	**0.50**	**0.90**	**0.55**	**0.25**	**0.88**	**0.28**	**0.12**	**0.96**	**0.12**	**0.08**	**0.55**	**0.15**

Three metrics are employed to gauge quality of result: 1) circuit speed, 2) area, and 3) energy consumption. For circuit speed, we consider the cycle latency, clock frequency and total execution time. Cycle latency refers to the number of clock cycles required for a complete execution of a benchmark. Clock frequency refers to the reciprocal of the post-routed critical path delay reported by Altera timing analysis. Total execution time is simply the cycle latency multiplied by the clock period. For area, we consider the number of used Cyclone II logic elements (LEs), memory bits, and 9x9 multipliers.

Energy is a key cost metric, as it directly impacts electricity costs, as well as influences battery life in mobile settings. To measure energy, we use Altera's PowerPlay power analyzer tool, applied to the routed design. We gather switching activity data for each benchmark through a post-route full delay simulation with Mentor Graphics' ModelSim. ModelSim produces a VCD (value change dump) file containing activity data for each design signal. PowerPlay reads the VCD to produce a power estimate for each design. To compute the total energy consumed by a benchmark for its computational work, we multiply the average core dynamic power reported by PowerPlay with the benchmark's total execution time.

5.1. Results

Table IV presents speed performance results for all circuits and flows. Three data columns are given for each flow: *Cycles* contains the latency in number of clock cycles; *Freq* presents the post-routed critical path delay in MHz; *Time* gives the total executation time in μS ($Cycles/Freq$). The flows are presented in the order specified above, from pure software on the left, to pure hardware on the right. The second last row of the table contains geometric mean results for each column. The *dhrystone* benchmark was excluded from the geomean calculations, as eXCite was not able to compile this benchmark. The last row of the table presents the ratio of the geomean relative to the software flow (*MIPS-SW*).

Beginning with the *MIPS-SW* flow, the data in Table IV indicates that the processor runs at 74 MHz on the Cyclone II and the benchmarks take between 6.7K-29M cycles to complete their execution. In terms of program execution time, this corresponds to a range of 92-401K μS[4]. In the *LegUp-Hybrid2* flow, where the second most compute-intensive function (and its descendants) is implemented as a hardware accelerator, the number of cycles needed for execution is reduced by 50% compared with software, on average. The *Hybrid2* circuits run at 10% lower frequency than the processor, on average. Overall, *LegUp-Hybrid2* provides a 45% (1.8×) speed-up in program execution time vs. software (*MIPS-SW*). Moving onto the *LegUp-Hybrid1* flow, which represents additional computations in hardware, Table IV

[4]As a comparison, we also ran the benchmarks on the Altera NIOS II/f (fast) soft processor and found the NIOS II performance to be about twice as fast as Tiger MIPS. Note, however, that NIOS II is not open source, has a 6-stage pipeline and is specially tuned for Altera devices, whereas, Tiger MIPS has a 5-stage pipeline and is not optimized for any particular FPGA device architecture.

shows that cycle latency is 75% lower than software alone. However, clock speed is 12% worse for this flow, which when combined with latency, results in a 72% reduction in program execution time vs. software (a 3.6× speed-up over software). Looking broadly at the data for *MIPS-SW*, *LegUp-Hybrid1* and *LegUp-Hybrid2*, we observe a trend: execution time decreases substantially as more computations are mapped to hardware. Note that the MIPS processor would certainly run at a higher clock speed on a 40/45 *n*m FPGA, e.g. Stratix IV, however the accelerators would also speed-up commensurately.

The two right-most flows in Table IV correspond to pure hardware implementations. Observe that benchmark programs mapped using the *LegUp-HW* flow require just 12% of the clock cycles of the software implementations, on average, yet they run at about the same speed in MHz. When benchmarks are mapped using *eXCite-HW*, even fewer clock cycles are required to complete their execution – just 8% of that required for software implementations. However, implementations produced by eXCite run at 45% lower clock frequency than the MIPS processor, on average. LegUp produces heavily pipelined hardware implementations, whereas, we believe eXCite does more operation chaining, resulting in few computation cycles yet longer critical path delays. Considering total execution time of a benchmark, LegUp and eXCite offer similar results. *LegUp-HW* provides an 88% execution time improvement vs. software (8× speed-up); *eXCite-HW* provides an 85% improvement (6.7× speed-up). Both of the pure hardware implementations are a significant win over software. The most favorable LegUp results were for the *dfdiv* and *dfsin* benchmarks, for which the speed-up over pure software was over 30×. The benchmark execution times of LegUp implementations relative to eXCite are comparable, which bodes well for our framework and gives us assurance that it produces implementations of reasonable quality.

Observe that neither of the hybrid scenarios provide a performance win over pure hardware for these particular benchmark circuits. Moreover, none of the benchmarks use `C` language constructs that are unsupported by LegUp. Nevertheless, the hybrid scenarios do serve to demonstrate LegUp's ability to synthesize working systems that contain both hardware and software aspects.

It is worth highlighting a few anomalous results in Table IV. Comparing *LegUp-HW* with *eXCite-HW* for the benchmark *aes*, LegUp's implementation provides a nearly 5× improvement over eXCite in terms of execution time. Conversely, for the *motion* benchmark, LegUp's implementation requires nearly 4× more cycles than eXCite's implementation. We believe such differences lie in the extent of pipelining used by LegUp vs. eXCite, especially for arithmetic operations such as division. In LegUp, we pipeline arithmetic units to the maximum extent possible, leading to higher cycle latencies, and improved clock periods.

Area results are provided for each circuit in Table V. For each flow, three columns provide the number of Cyclone II logic elements (*LEs*), the number of memory bits used (*# bits*), as well as the number of 9x9 multipliers (*Mults*). As in the performance data above, the geometric mean and ratios relative to MIPS software alone are given in the last two rows of Table V. Observe that some columns contain a `0` for one or more circuits, invalidating the geomean calculation. To calculate the geomean for such columns, the `0`'s were taken to be `1`'s[5].

Beginning with the area of the MIPS processor, the data in Table V shows it requires 12.2K LEs, 226K memory bits, and 16 multipliers. The hybrid flows include *both* the MIPS processor, as well as custom hardware, and consequently, they consume considerably more area. When the *LegUp-Hybrid2* flow is used, the number of LEs, memory bits, and multipliers increase by 2.23×, 1.14×, and 2.68×, respectively, in Hybrid2 vs. the MIPS processor alone, on average. The *LegUp-Hybrid1* flow requires even more area: 2.75× LEs, 1.16× memory bits, and 3.18× multipliers vs. MIPS. Note that *link time* optimization in LLVM was disabled for the hybrid flows, as was necessary to preserve the integrity of the function

[5]This convention is used in life sciences studies.

Table V. Area results.

	MIPS-SW			LegUp-Hybrid2			LegUp-Hybrid1			LegUp-HW			eXCite-HW		
Benchmark	**LEs**	**# bits**	**Mults**	**LEs**	**# bits**	**Mults**	**LEs**	**# bits**	**Mults**	**LEs**	**# bits**	**Mults**	**LEs**	**# bits**	**Mults**
adpcm	12243	226009	16	25628	242944	152	46301	242944	300	22605	29120	300	16654	6572	28
aes	12243	226009	16	56042	244800	32	68031	245824	40	28490	38336	0	46562	18688	0
blowfish	12243	226009	16	25030	341888	16	31020	342752	16	15064	150816	0	31045	33944	0
dfadd	12243	226009	16	22544	233664	16	26148	233472	16	8881	17120	0	9416	0	0
dfdiv	12243	226009	16	28583	226009	46	36946	233472	78	20159	12416	62	9482	0	32
dfmul	12243	226009	16	16149	226009	48	20284	233472	48	4861	12032	32	4536	0	26
dfsin	12243	226009	16	34695	233472	78	54450	233632	116	38933	12864	100	22274	0	38
gsm	12243	226009	16	25148	232576	114	30808	233296	142	19131	11168	70	6114	3280	2
jpeg	12243	226009	16	46432	338096	252	64441	354544	254	46224	253936	172	30420	105278	20
mips	12243	226009	16	18857	230304	24	18857	230304	24	4479	4480	8	2260	3072	8
motion	12243	226009	16	28761	243104	16	18013	242880	16	13238	34752	0	20476	16384	0
sha	12243	226009	16	20382	359136	16	29754	359136	16	12483	134368	0	13684	3072	0
dhrystone	12243	226009	16	15220	226009	16	16310	226009	16	4985	82008	0	-	-	-
Geomean:	**12243**	**226009**	**16**	**27248**	**258526**	**43**	**33629**	**261260**	**51**	**15646**	**28822**	**12**	**13101**	**496**	**5**
Ratio:	**1**	**1**	**1**	**2.23**	**1.14**	**2.68**	**2.75**	**1.16**	**3.18**	**1.28**	**0.13**	**0.72**	**1.07**	**0.00**	**0.32**

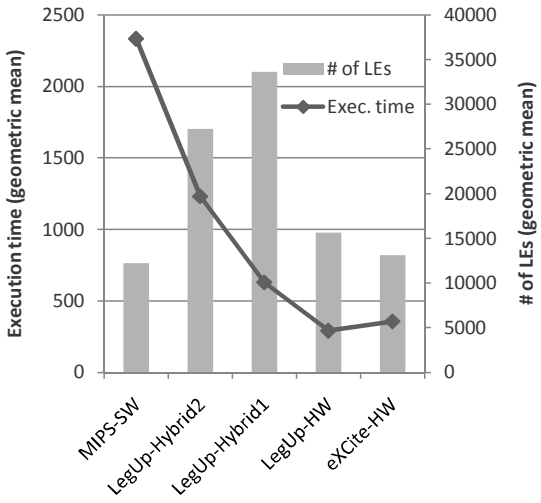

Fig. 9. Performance and area results (performance in μS).

boundaries[6]. However, link time optimization was enabled for the *MIPS-SW* and *LegUp-HW* flows, permitting greater compiler optimization for such flows, possibly improving area and speed.

Turning to the pure hardware flows in Table V, the *LegUp-HW* flow implementations require 28% more LEs than the MIPS processor on average; the *eXCite-HW* implementations require 7% more LEs than the processor. In other words, on the key area metric of the number of LEs, LegUp implementations require 19% more LEs than eXCite, on average. We consider the results to be quite encouraging, given that this is the initial release of an open source academic HLS tool. In terms of memory bits, both the *LegUp-HW* flow and the *eXCite-HW* flow require much fewer memory bits than the MIPS processor alone. For the benchmarks that require embedded multipliers, the *LegUp-HW* implementations use more multipliers than the *eXCite-HW* implementations, which we believe is due to more extensive multiplier sharing in the binding phase of eXCite.

Figure 9 summarizes the speed and area results. The left vertical axis represents geometric mean execution time; the right axis represents area (number of LEs). Observe that execution

[6]Link time optimization permits code optimization across compilation modules.

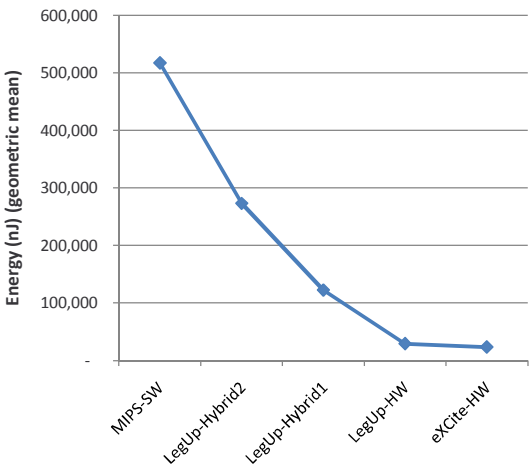

Fig. 10. Energy results.

time drops as more computations are implemented in hardware. While the data shows that pure hardware implementations offer superior speed performance to pure software or hybrid implementations, the plot demonstrates LegUp's usefulness as a tool for exploring the hardware/software co-design space. One can multiply the delay and area values to produce an *area-delay product*. On such a metric, *LegUp-HW* and *eXCite-HW* are nearly identical (∼4.6M μS-LEs vs. ∼4.7M μS-LEs) – *LegUp-HW* requires more LEs vs. *eXCite-HW*, however, it offers better speed, producing a roughly equivalent area-delay product. The area-delay product parity with eXCite gives us further confidence that the HLS results produced by LegUp are competitive with commercial tools.

Figure 10 presents the geometric mean energy results for each flow. The energy results bear similarity to the trends observed for execution time, though the trends here are even more pronouced. Energy is reduced drastically as computations are increasingly implemented in hardware vs. software. The *LegUp-Hybrid2* and *LegUp-Hybrid1* flows use 47% and 76% less energy than the *MIPS-SW* flow, respectively, representing 1.9× and 4.2× energy reductions. The pure hardware flows are even more promising from the energy standpoint. With *LegUp-HW*, the benchmarks use 94% less energy than if they are implemented with the *MIPS-SW* flow (an 18× reduction). The eXCite results are similar. Pure hardware benchmark implementations produced by eXCite use over 95% less energy than software implementations (a 22× reduction). The energy results are promising, especially since energy was not a specific focus of our initial release.

6. EXTENSIBILITY OF LEGUP: CASE STUDIES

We now move onto illustrating the extensibility of the LegUp framework in several ways: 1) we describe our recent work on supporting an additional FPGA architecture, 2) we describe our experiences with incorporating a state-of-the-art scheduling algorithm, and 3) we show a simple example that utilizes multiple parallel accelerators. Our intent is to demonstrate LegUp's utility as an open source tool that can be tailored by the researcher to meet their particular needs.

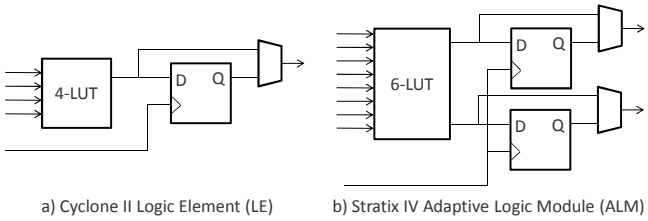

Fig. 11. Cyclone II and Stratix IV logic element architectures.

6.1. Targeting a Different FPGA Device

As described above, the LegUp targets the Altera DE2 board with the Cyclone II FPGA. We expect, however, that some users will wish to target other platforms and we therefore investigated the ease with which the tool can be modified to target a different FPGA. We chose Altera's 40*n*m Stratix IV family as the alternative platform [Altera, Corp. 2011]. Stratix IV has a considerably different logic element architecture than Cyclone II. Cyclone II uses logic elements (LEs) with 4-input LUTs to implement combinational logic functions, whereas Stratix IV uses an adaptive logic module (ALM). An ALM is a dual-output 6-LUT that receives eight inputs from the interconnection fabric. The ALM can implement any 6-input function, any two 4-input functions, a 3- and 5-input function, and several other combinations. Each of the two LUT outputs has a flip-flop that can be optionally bypassed. The Cyclone II and Stratix IV logic elements are illustrated in Fig. 11.

LegUp HLS depends on the target FPGA architecture in two ways: 1) the area and delay models for each of the LLVM operations (including the latency of each operator, i.e. the number of clock cycles each operator takes to complete), 2) architecture-specific aspects of the Verilog generation. For #2, LegUp's Verilog is vendor agnostic except in two respects: block RAM and divider instantiation. Block RAMs are instances of Altera's `ALTSYNCRAM` megafunction and division is done with the `LPM_DIVIDE` megafunction. Thus, very little has to be changed to target another vendor's FPGAs or to use the tool as a front-end to the open source VTR (Verilog-to-routing) FPGA CAD flow being developed at the University of Toronto [VTR 2011]. VTR accepts Verilog RTL as input.

As mentioned in Section 4.1.2, the speed and area of each LLVM operator is stored in a `Tcl` file that is read by LegUp. The area/speed data must be updated if LegUp is to produce good results for a particular target device, as the data is used within the scheduling and binding algorithms. To ease migration to alternative FPGA devices, the LegUp installation includes a set of Verilog modules that implement each LLVM operator in isolation. One can compile each of the modules to the particular FPGA device being targeted to populate the `Tcl` file. For Altera FPGAs specifically, a set of PERL scripts is provided that automatically compile the single-operator modules using Altera commercial tools and then parse the results to retrieve the area and speed data. We executed the scripts to gather Stratix IV area and speed data for each LLVM operator and then manually modified the `Tcl` script. Gathering the speed/area data and modifying the `Tcl` script took a day.

Table VI gives speed and area results for Stratix IV; the results for Cyclone II are repeated in the table for convenience. Speed and area results are shown on the left and right sides of the table, respectively. Looking first at the speed data, we see that as expected, all circuits run considerably faster in Stratix IV vs. Cyclone II – the speed-up ratio is 2.4×, which we expect is partly due to the more advanced technology node, and partly due to the Stratix IV architecture. Area values are given in LEs and ALMs for Cyclone II and Stratix IV, respectively. On average, about 60% fewer ALMs are needed to implement circuits in Stratix IV vs. LEs in Cyclone II, owing to Stratix IV's larger dual-output LUTs.

Table VI. Stratix IV Speed and Area Results.

	Performance (MHz)		Area (LEs, ALMs)	
Benchmark	**Cyclone II**	**Stratix IV**	**Cyclone II**	**Stratix IV**
adpcm	45.79	115.51	22605	9993
aes	60.72	135.81	28490	9088
blowfish	65.41	198.77	15064	7560
dfadd	124.05	247.28	8881	4196
dfdiv	74.72	159.85	20159	7540
dfmul	85.62	248.32	4861	1965
dfsin	62.64	143.97	38933	15044
gsm	58.93	143.43	19131	6662
jpeg	47.09	82.69	46224	24883
mips	90.09	223.21	4479	2541
motion	91.79	248.82	13238	2546
sha	86.93	212.9	12483	4889
dhrystone	85.38	182.32	4985	2735
Geomean:	**72.54**	**171.69**	**14328.0**	**5840.8**

Porting LegUp to an alternative FPGA device for pure hardware HLS is straightforward, however, supporting the hybrid processor/accelerator scenario on a non-Altera device is more involved. In particular, the Tiger MIPS processor makes use of Altera megafunctions for memory, division and multiplication. The megafunctions would need to be changed to reference the corresponding modules for the alternate FPGA vendor. Moreover, as described in Section 3.2, the LegUp hybrid platform uses the Altera Avalon interface for processor/accelerator communication. If a Xilinx FPGA were targeted, processor/accelerator system generation and communication would need to be modified to use the Xilinx EDK tool and PLB bus [PLB 2011]. The PLB and Avalon interfaces are quite similar however, as both are memory-mapped master/slave bus interfaces. We therefore see no significant barriers that would prevent LegUp from targeting a Xilinx device.

6.2. Implementing a New Scheduling Algorithm

We implemented a new scheduling algorithm in LegUp based on the SDC (system of difference constraints) formulation, as described in [Cong and Zhang 2006], and used in xPilot [Cong et al. 2006]. The idea is to formulate the scheduling problem as a linear program (LP) that can be solved with a standard solver (we used lpsolve [LPS 2011]). In SDC scheduling, each operation is assigned a variable that, after solving, will hold the clock cycle in which the operation in scheduled. Consider two operations, $op1$ and $op2$, and let the variable c_{op1} represent the cycle in which $op1$ is to be scheduled, and c_{op2} the cycle in which $op2$ is to be scheduled. Assume further that $op2$ *depends on* $op1$, then the following constraint is added to the LP formulation: $c_{op1} \geq c_{op2}$ or equivalently: $c_{op1} - c_{op2} \geq 0$: a *difference constraint*.

Clock period constraints can also be incorporated into SDC scheduling. Let P be the target clock period and let C represent a chain of any N dependant combinational operations in the dataflow graph: $C = op1 \rightarrow op2 \rightarrow ... \rightarrow opN$. Assume that T represents the total estimated combinational delay of the chain of N operations – computed by summing the delays of each operator (with the operators characterized as described in Section 4.1.3). We can add the following timing constraint to the LP: $c_{opN} - c_{op1} \geq \lceil T/P \rceil - 1$. This difference constraint requires that the cycle assignment for opN be at least $\lceil T/P \rceil - 1$ cycles later than the cycle in which $op1$ is scheduled. Such constraints control the extent to which operations can be chained together in a clock cycle. Chaining is permitted such that the target clock period P is met. This provides the basic picture of SDC scheduling and the reader is referred to [Cong and Zhang 2006] for complete details of the formulation and how other types of constraints can be included.

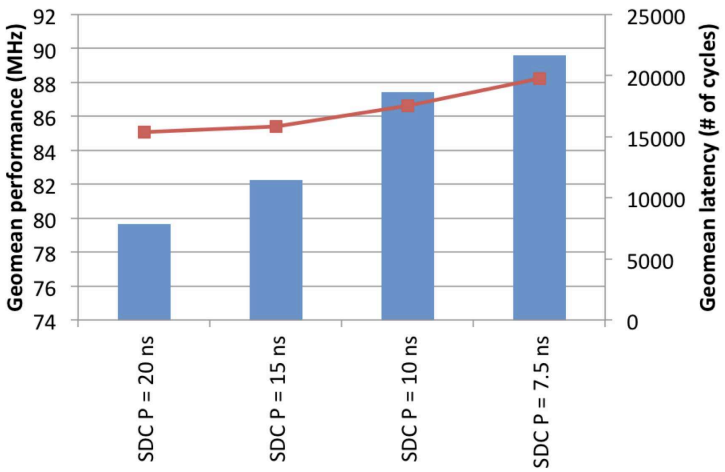

Fig. 12. SDC scheduling results for Cyclone II with various clock period constraints (bars represent performance in MHz; the line represents latency in clock cycles.

The LegUp implementation has a *scheduler DAG* object that, essentially, annotates each LLVM instruction with data relevant to its scheduling: its combinational delay, as characterized in the target FPGA, and the instructions on which it depends. The scheduler DAG object can be viewed as an overlay on the dataflow graph with scheduling-specific information. The object contains all of the information needed for us to generate the SDC LP formulation. After solving the LP, we deposit the cycle assignment for each instruction into another LegUp data structure called the *scheduler mapping*. For each LLVM instruction, the mapping holds the scheduled cycle number. Following scheduling, FSM generation accesses the mapping object to construct the FSM.

Fig. 12 shows SDC scheduling results for Cyclone II, demonstrating the impact of running SDC with different clock period constraints. The left axis (bar) gives the geometric mean post-routed clock frequency across the 12 CHStone circuits and dhrystone; the right axis (line) gives the geometric mean latency (# of clock cycles to execute). The four datapoints show SDC scheduling results for clock period constraints of 20, 15, 10, and 7.5 ns, respectively. Observe that circuit clock frequency increases as P is decreased, which demonstrates the effectiveness of SDC, as well as provides confidence in our operator speed characterization. Note that P is a *minimum* clock period constraint – no effort is made to actually slow circuits down. Hence, for the $P = 20$ ns datapoint, the circuits run considerably faster than 50 MHz. As P is decreased, the circuits are more heavily pipelined and take larger numbers of cycles to execute.

SDC scheduling will be made LegUp's default scheduling algorithm in a subsequent release.

6.3. Parallel Accelerators

As a last case study, we demonstrate the capability of LegUp to synthesize multi-accelerator systems. As a proof-of-concept application, we use array addition for four 1000-element arrays. Three parallelization scenarios were evaluated: 1) pure software with the MIPS processor performing all of the work, 2) a single accelerator, called by the processor, performing each of the four array additions sequentially, and, 3) four accelerators, operating in parallel, with each accelerator performing the addition for one of the four arrays. In the multi-accelerator case, the processor signals each accelerator to start its work and polls until all four have completed. We found that a single acclerator doing all of the work sequentially

provides a 5.2× speedup over the pure software case. Using four parallel accelerators yields a 3.7× speedup vs. using a single accelerator. While this is a simple application, with no potential cache coherency issues, it serves to illustrate that concurrently running accelerators are feasible with LegUp – a topic we plan to explore further in future work.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced LegUp – a new high-level synthesis tool that compiles a standard `C` program to a hybrid processor/accelerator architecture comprising a MIPS processor and custom accelerators communicating through a standard on-chip interface. Using LegUp, one can explore the hardware/software design space, where some portions of a program run on a processor, and others are implemented as custom hardware circuits. As compared with software running on a MIPS soft processor, pure hardware implementations produced by LegUp HLS execute 8× faster and use 18× less energy on a Altera Cyclone II FPGA. LegUp's hardware implementations are competitive with those produced by a commercial HLS tool, both in benchmark execution time and in area-delay product. LegUp, along with its suite of benchmark C programs, is a powerful open source platform for HLS research that we expect will enable a variety of research advances in hardware synthesis, as well as in hardware/software co-design. LegUp is available for download at: `http://www.legup.org`.

We are currently using the LegUp framework to explore several new directions towards improving computational throughput. First, we are investigating the benefits of using multiple clock domains, where each processor and accelerator can operate at its maximum speed and communication between modules occurs across clock domains (the Altera Avalon interface can support this). Second, we are implementing *loop pipelining* within our scheduler, wherein a loop iteration can commence execution prior to the completion of the previous iteration (a loop iteration's instructions can execute as long as their operands have been computed). Lastly, although we are already seeing significant energy benefits of computing in hardware vs. software, we believe that much more can be done on this front through the incorporation of energy-driven scheduling and FSM generation. Lastly, we are exploring the performance benefits of concurrent processor/accelerator execution for both data parallel and task parallel applications, as well as for multi-program workloads.

Our long-term vision is to fully automate the flow in Figure 1, thereby creating a *self-accelerating adaptive processor* in which profiling, hardware synthesis and acceleration happen transparently without user awareness.

Acknowledgements

The authors thank Dr. Tedd Hadley from Y Explorations for providing the eXCite tool used in the experimental study. The authors gratefully acknowledge the comments of the anonymous reviewers that have significantly improved the manuscript.

REFERENCES

2011. CoreConnect, Xilinx, Inc. *http://www.xilinx.com/support/documentation/ipembedprocess_coreconnect.htm.*

2011. lp_solve linear programming solver. *http://lpsolve.sourceforge.net/5.5/.*

2011. VTR – the Verilog-to-routing project for FPGAs. *http://www.eecg.toronto.edu/vtr/.*

Aldham, M., Anderson, J., Brown, S., and Canis, A. 2011. Low-cost harware profiling of run-time and energy in FPGA embedded processors. In *IEEE Int'l Conference on Application-specific Systems, Architecture and Processors (ASAP).* 61–68.

Altera, Corp. 2009. *Nios II C2H Compiler User Guide.* Altera, Corp., San Jose, CA.

Altera, Corp. 2010. *Avalon Interface Specification.* Altera, Corp., San Jose, CA.

Altera, Corp. 2011. *Stratix IV FPGA Family Data Sheet.* Altera, Corp., San Jose, CA.

AutoESL. *AutoESL Design Technologies, Inc. (http://www.autoesl.com).* AutoESL.

Betz, V. and Rose, J. 1997. VPR: A new packing, placement and routing tool for FPGA research. In *Int'l Workshop on Field Programmable Logic and Applications.* 213–222.

Cadence 2010. *Cadence C-to-Silicon Compiler (http://www.cadence.com/products/sd/silicon_compiler)*. Cadence.

Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Anderson, J., Brown, S., and Czajkowski, T. 2011. LegUp: High-level synthesis for FPGA-based processor/accelerator systems. In *ACM Int'l Symposium on Field Programmable Gate Arrays*. 33–36.

CebaTech 2010. *CebaTech The software to silicon company (http://www.cebatech.com)*. CebaTech.

Chen, D. and Cong, J. 2004. Register binding and port assignment for multiplexer optimization. In *IEEE/ACM Asia and South Pacific Design Automation Conference*. 68–73.

Cong, J., Fan, Y., Han, G., Jiang, W., and Zhang, Z. 2006. Platform-based behavior-level and system-level synthesis. In *IEEE Int'l System-on-Chip Conference*. 199–202.

Cong, J. and Zhang, Z. 2006. An efficient and versatile scheduling algorithm based on SDC formulation. In *IEEE/ACM Design Automation Conference*. 433–438.

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

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

Coussy, P., Lhairech-Lebreton, G., Heller, D., and Martin, E. 2010. GAUT – a free and open source high-level synthesis tool. In *IEEE Design Automation and Test in Europe – University Booth*.

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

Forte 2010. *Forte Design Systems The high level design company (http://www.forteds.com/products/cynthesizer.asp)*. Forte.

Gajski, D. and et. al. Editors. 1992. *High-Level Synthesis - Introduction to Chip and System Design*. Kulwer Academic Publishers.

Hara, Y., Tomiyama, H., Honda, S., and Takada, H. 2009. Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis. *Journal of Information Processing 17*, 242–254.

Henkel, J. 2003. Closing the SoC design gap. *IEEE Computer 36*, 119–121.

Huang, C., Che, Y., Lin, Y., and Hsu, Y.

Huang, S., Hormati, A., Bacon, D., and Rabbah, R. 2008. Liquid Metal: Object-oriented programming across the hardware/software boundary. In *ACM European conference on Object-Oriented Programming*. 76–103.

Impulse 2010. *Impulse CoDeveloper – Impulse accelerated technologies (http://www.impulseaccelerated.com)*. Impulse.

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

LLVM 2010. *The LLVM Compiler Infrastructure Project (http://www.llvm.org)*. LLVM.

Luu, J., Redmond, K., Lo, W., Chow, P., Lilge, L., and Rose, J. 2009. FPGA-based monte carlo computation of light absorption for photodynamic cancer therapy. In *IEEE Int'l Symposium on Field-Programmable Custom Computing Machines*. 157–164.

Mentor Graphics 2010. *Mentor Graphics (http://www.mentor.com/products/esl/high_level_synthesis)*. Mentor Graphics.

Mishchenko, A., Chatterjee, S., and Brayton, R. 2006. DAG-aware AIG rewriting: A fresh look at combinational logic synthesis. In *IEEE/ACM Design Automation Conference*. 532–536.

Pothineni, N., Brisk, P., Ienne, P., Kumar, A., and Paul, K. 2010. A high-level synthesis flow for custom instruction set extensions for application-specific processors. In *ACM/IEEE Asia and South Pacific Design Automation Conference*. 707–712.

Pozzi, L., Atasu, K., and Ienne, P. 2006. Exact and approximate algorithms for the extension of embedded processor instruction sets. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 25,* 7, 1209–1229.

Putnam, A., Bennett, D., Dellinger, E., Mason, J., Sundararajan, P., and Eggers, S. 2008. CHiMPS: A C-level compilation flow for hybrid CPU-FPGA architectures. In *IEEE Int'l Conference on Field Programmable Logic and Applications*. 173–178.

Stitt, G. and Vahid, F. 2007. Binary synthesis. *ACM Transactions on Design Automation of Electronic Systems 12,* 3.

Sun, F., Raghunathan, A., Ravi, S., and Jha, N. 2004. Custom-instruction synthesis for extensible-processor platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 23,* 7, 216–228.

Tripp, J., Gokhale, M., and Peterson, K. 2007. Trident: From high-level language to hardware circuitry. *IEEE Computer 40,* 3, 28–37.

United States Bureau of Labor Statistics 2010. *Occupational Outlook Handbook 2010-2011 Edition*. United States Bureau of Labor Statistics.

University of Cambridge 2010. *The Tiger MIPS processor (http://www.cl.cam.ac.uk/teaching/0910/ECAD+Arch/mips.html)*. University of Cambridge.

Vahid, F., Stitt, G., and R., L. 2008. Warp processing: Dynamic translation of binaries to FPGA circuits. *IEEE Computer 41,* 7, 40–46.

Villarreal, J., Park, A., Najjar, W., and Halstead, R. 2010. Designing modular hardware accelerators in C with ROCCC 2.0. In *IEEE Int'l Symposium on Field-Programmable Custom Computing Machines*. 127–134.

Wayne Marx, V. A. 2008. FPGAs Are Everywhere - In Design, Test & Control. *RTC Magazine*.

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