Automating the Design of Processor/Accelerator Embedded Systems with LegUp High-Level Synthesis

Blair Fort, Andrew Canis, Jongsok Choi, Nazanin Calagar, Ruolong Lian, Stefan Hadjis, Yu Ting Chen,
Mathew Hall, Bain Syrowik, Tomasz Czajkowski*, Stephen Brown, Jason Anderson
Dept. of Electrical and Computer Engineering, University of Toronto, Toronto, Ontario, Canada
*Altera Toronto Technology Centre, Toronto, Ontario, Canada
Email: legup@eecg.toronto.edu

***Abstract*—LegUp [1] is an open-source high-level synthesis (HLS) tool that accepts a C program as input and automatically synthesizes it into a hybrid system. The hybrid system comprises an embedded processor and custom accelerators that realize user-designated compute-intensive parts of the program with improved throughput and energy efficiency. In this paper, we overview the LegUp framework and describe several recent developments: 1) support for an embedded ARM processor, as is available on Altera's recently released SoC FPGA; 2) HLS support for software parallelization schemes – pthreads and OpenMP; 3) enhancements to LegUp's core HLS algorithms that raise the quality of the auto-generated hardware; and, 4) a preliminary debugging and verification framework providing C source-level debugging of HLS hardware. Since its first release in 2011, LegUp has been downloaded over 1000 times by groups around the world, providing a powerful platform for new research in high-level synthesis algorithms and embedded systems design.**

I. Introduction

Embedded systems typically include a processor in combination with some dedicated hardware accelerators. The processor is used to run software programs that solve relatively simple tasks needed in the embedded system, and the hardware accelerators are used for compute- or energy-intensive tasks for which a processor is not efficient. It is well known (e.g. [2]) that hardware implementations of algorithms can often result in an order of magnitude, or more, improvement over the same algorithms when executed in software on a processor. These improvements may be in computational speeds, energy usage, or both. A significant challenge, however, is that hardware design often requires significantly more effort and expertise in comparison to software development, and the needed expertise is not readily available.

High-level synthesis (HLS) refers to the automated synthesis of a hardware circuit from a software program, with the promise of offering hardware's advantages to those with only software skills. While the concept of HLS is not new [3] it has only recently gained traction in the industry as a viable alternative to hardware design using hardware description languages (HDLs). Possible reasons for an improved popularity of HLS include: 1) HLS tools have become more readily available at lower costs, and produce better results than previous tools, 2) hardware platforms, such as FPGAs, have become large enough to justify the need for higher-level design approaches, due to the inherent difficulty of developing very large circuits by using hardware description languages, and 3) shorter lifecycles for electronic products, emphasizing the need to produce products quickly.

Field-programmable gate arrays (FPGAs) are computer chips that can be programmed by the end-user to implement *any* digital circuit. FPGAs can be thought of as "configurable" computer hardware, making them an ideal platform to realize HLS-synthesized accelerators. While there is naturally a gap in some metrics (area, speed and power) between HLS hardware and hand-crafted hardware, the "cost" of that gap is more tolerable in the FPGA context relative to custom silicon. For this reason, we believe FPGAs will be the technology through which HLS enters the mainstream of HW design. Indeed, the two largest FPGA vendors have invested heavily in HLS technologies in recent years [4], [5].

In this paper, we describe LegUp [1], an open-source HLS tool being developed at the University of Toronto. LegUp synthesizes a `C`-language program into an FPGA-based hybrid system comprising a processor and one or more accelerators that, together with the processor, implement the program with improved area efficiency, speed and power vs. the use of software alone. LegUp can also be used in a more traditional HLS flow, to synthesize an entire program to hardware. Active research on LegUp can be categorized into four thrusts overviewed in subsequent sections of this paper: 1) Support for an ARM hard processor, as available on the recently released Altera Cyclone V SoC [6]; 2) improving performance of the syntesized HW by increasing parallelism using approaches such as better scheduling; 3) improvements to the synthesized memory architecture and support for HW multi-cycling to raise clock frequency; and 4) a debugging framework that provides `gdb`-like debugging functionality for HLS hardware.

II. System Overview

LegUp leverages the state-of-the-art LLVM (low-level virtual machine) compiler framework for high-level language parsing and its standard compiler optimizations [7]. We integrated the hardware synthesis process as back-end compiler passes within LLVM.

Fig. 1 illustrates the LegUp design flow. The designer starts with the software implementation of the application in standard `C` code. At step ①, the `C` program is compiled to a binary executable targeting a processor. While the program executes on the processor [8] at step ②, a hardware profiler [9] collects the profiling data and identifies the critical sections of the program that can benefit most from hardware acceleration. At step ③, the user, based on the profiling data, marks specific functions to be synthesized into hardware accelerators. At

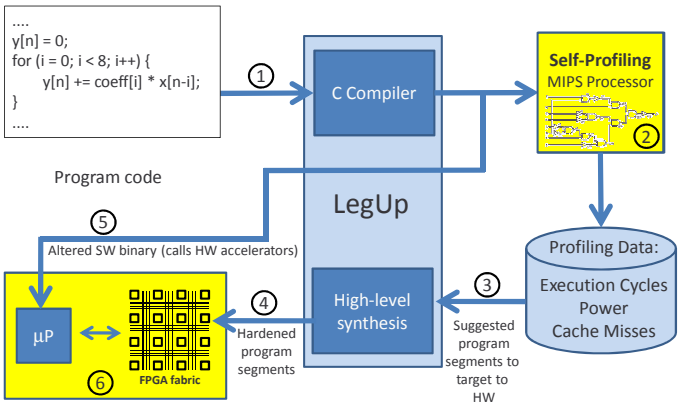


Fig. 1. Design flow with LegUp [1].

step ④, LegUp's high-level synthesis engine is invoked to synthesize these functions into hardware accelerators described in Verilog code. Next, in step ⑤, the `C` source is re-compiled with the accelerated functions replaced by wrapper functions, which are used to invoke hardware accelerators. Lastly, the hybrid processor/accelerator system executes on the FPGA in step ⑥.

A. Integrating Custom Verilog

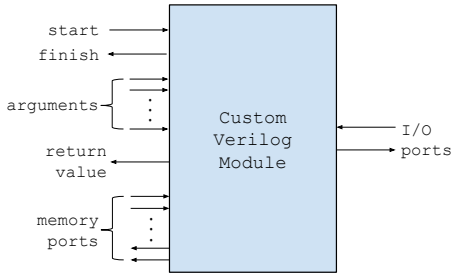


Fig. 2. Sample of custom Verilog module interface.

In many embedded systems, accelerators need to communicate with off-chip components through I/O ports, following certain interface protocols such as JTAG, PCIe or USB. Such interfaces have specific cycle-by-cycle protocol specifications, and therefore we would prefer to describe the interface behavior directly in a hardware description language. LegUp provides a convenient solution to integrate custom Verilog modules as adaptors for software functions to access standard I/O ports. Custom modules must adhere to a specific interface so they can be used by LegUp-generated hardware, as shown in Fig. 2. On the left side, we show the interface to the LegUp generated hardware, or *caller* module, while the right side connects to external I/O ports that are specified in a LegUp configuration file. The HLS-generated caller module provides function arguments through the `arguments` ports that match the function prototype in `C`, and then triggers the `start` signal to launch the custom module. The `memory ports` allow the custom module to read/write the memory at addresses usually provided as function arguments. Upon completion, the custom module sets the data on the `return value` and asserts the `finish` signal to notify the caller module. Designers can also use custom modules to stitch in hand-tuned hardware for DSP algorithms that can be used by the LegUp auto-generated hardware.

III. LegUp Accelerators with an ARM Processor

As commercial FPGA devices have evolved, there has been an increasing tendency to include dedicated hardware blocks, such as memories, DSP units, and, most recently, embedded processors. Both major FPGA vendors, Altera and Xilinx, have seen a need to provide these processors in their devices. For many years now, Altera and Xilinx have provided soft processors, called Nios II and Microblaze, respectively, which are implemented in the FPGA fabric. The performance of these processors is limited and highly dependent on the FPGA family being used. Recently, the FPGA vendors have included a hard processor tightly coupled with the FPGA fabric. Specifically, they chose a dual-core ARM Cortex-A9. The hard ARM processor provides an order of magnitude better performance versus soft processors [10]. To leverage the benefits of the ARM processor, we have added the ability to target it as an alternative to the soft-core processor previously available in LegUp.

In the remainder of this section, we will discuss: 1) the architecture of the hard processor system in Altera SoC devices; 2) modifications and issues for supporting the ARM processor within the LegUp design flow; and 3) experimental results.

A. System Architecture

The ARM-based Hard Processor System (HPS) on Altera's SoC devices, as seen in Fig. 3, contains processor cores, a memory controller and peripherals. The HPS has either one or two ARM Cortex-A9 processors. Each processor has independent 32-KB L1 instruction and data caches. Cache coherency between the separate L1 caches in the two processors is maintained by the snoop control unit (SCU). There is also one 512-KB shared L2 cache. The L2 cache is connected to a DDR2/DDR3 SDRAM memory controller and the L3 interconnect. The L3 interconnect enables communication with the memory-mapped peripherals of the HPS, such as timers, ethernet, a 64-KB RAM, etc. Also, the L3 interconnect enables communication to and from circuits implemented in the FPGA fabric. Circuits in the FPGA can be slaves and/or masters attached to the L3 interconnect via the FPGA bridge. Also, the FPGA can directly communicate with the SDRAM memory controller.

The memory layout, as seen by the processor, is divided into three main regions: SDRAM, FPGA slaves and the HPS peripheral region. The lower 3 GBs of addressable space is allocated to the SDRAM, and 960 MBs of space is allocated for FPGA slaves. The rest is for the HPS peripherals. The FPGA's view is slightly different. Addresses between 2 GB (`0x80000000`) and 3 GB (`0xBFFFFFFF`) are mapped to the accelerator coherency port (ACP). The ACP allows FPGA peripherals to access data in a cache-coherent manner, by routing transfers through the SCU and L2 cache. As the ACPs addressable space is limited, it can only access 1 GB at a time. However, the ACP contains an address translation mechanism to allow cache coherent access to the full addressable memory space.

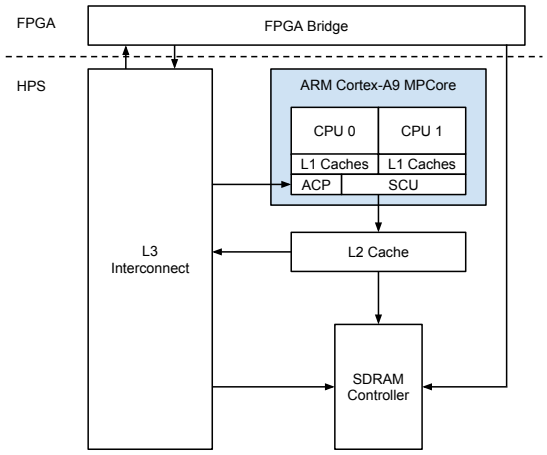


Fig. 3. Altera's Cyclone V SoC Architecture.

B. Design Flow

The design flow for using LegUp with the ARM processor is similar to our description in Section II for using LegUp with a soft processor. The main difference is that `C` programs are now compiled to an ARM executable. Also, in step ②, we profile the program on the ARM processor, while collecting the same runtime statistics as we did when using the soft processor. More specific differences between targeting the ARM processor vs. the soft processor include accessing global variables, using the cache and HPS specific initialization.

LegUp determines whether variables used within functions being accelerated are local or global. Local variables are stored in on-chip memory that is instantiated within the hardware accelerators. Global variables are stored in memory accessible by both the processor and the hardware accelerators. To access global variables, the accelerators have a memory controller, which is connected to the L3 interconnect through the FPGA bridge. During execution, the memory controller determines whether a data access is local or global by checking the address. For the soft processor, the addresses between `0x00800000` and `0x00FFFFFF` were allocated to be global data. For simplicity, the same selection was made for the ARM processor. Consequently, LegUp compiles programs to have a starting address of `0x00800000` and we initialize the stack pointer to `0x01000000`.

The soft-core processor does not have an internal data cache. Rather, we instantiated one in the FPGA fabric. As the data cache was a separate component in the system, it was trivial to connect the accelerators to the cache. To access the cache in the Altera HPS, memory transactions from the FPGA must go through the ACP. This requires two steps. Firstly, all addresses used by hardware accelerators have to be shifted up by `0x80000000`. Secondly, the ACP must translate the addresses back to the correct location. Conveniently, the ACP's default settings provide the correct translation.

Upon resetting the HPS, the data cache and the FPGA bridge are disabled. To enable the data cache, the processor must enable the MMU and set a one-to-one mapping in its address translation table. LegUp automatically adds code to enable both the data cache and the FPGA bridge.

C. Experimental Results

We evaluated the ARM processor design flow using the Altera DE1-SoC [11] board, which contains a Cyclone V SoC device. We performed experiments using a benchmark that computes the Mandelbrot set for an image of 128×128 pixels. Table I shows the execution time when the benchmark is run in pure software (the SW row) on the ARM processor. The table also contains the execution time when the main computational function is run as a single hardware accelerator attached to the ARM processor (1 hardware thread). In this case, the hardware executes the Mandelbrot set approximately 11% faster than the pure software case.

TABLE I
ACCELERATORS WITH AN ARM PROCESSOR.

	Wall-clock time (ms)	**Cycles**	**FMax (MHz)**	**Speedup**	**ALMs**
SW	28.03	-	-	-	-
1	25.28	3084718	122	1.11×	2484 (8%)
2	15.36	1550883	101	1.83×	4439 (14%)
4	7.83	775603	99	3.58×	7992 (25%)
8	4.91	388223	79	5.70×	15959 (50%)

LegUp also has the ability to convert pthreads into multiple accelerators, which is discussed in Section IV. Table I shows the execution time, number of cycles and FMax for the Mandelbrot set, when we parallelize the benchmark across 2, 4 and 8 hardware accelerators. As we add more accelerators, the cycles required decreases linearly; however, the FMax also decreases and hence the overall speedup does not improve linearly. The best speedup we achieved was 5.70× with 8 accelerators.

IV. Extracting Parallelism from Sequential Code

A key challenge in HLS is to exploit the spatial parallelism available on the FPGA, as it is difficult to automatically extract parallelism from sequential `C` code. LegUp addresses this challenge by providing support for thread-level parallelism, where parallel software threads are compiled to concurrent hardware accelerators, as well as loop-level parallelism, where loop pipelining is used to create pipelined hardware that can execute multiple loop iterations concurrently.

A. Synthesizing Software Threads into Parallel Hardware

LegUp provides support for using Pthreads and OpenMP for the specification of parallelism [12]. Parallelism described in the software code is automatically synthesized into parallel hardware accelerators that perform the corresponding computations concurrently. 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 processor. The processor invokes accelerators and retrieves their return values by using wrapper functions, which replace the original software versions of the parallel code. Wrapper functions perform memory-mapped writes/reads over the on-chip bus to send function arguments to the parallel accelerators, start the accelerators, poll to check if they are done, then retrieve any return values. Writing deterministic parallel software often requires the use of synchronization constructs

TABLE II
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

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.

Table II shows a list of Pthreads and OpenMP library functions which are currently supported in our framework. Note that all of the original calls to OpenMP/Pthreads functions are automatically replaced with corresponding functions 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*.

We also allow *nested parallelism* – threads forking threads. Consider the case of there being multiple functions executed in parallel with Pthreads – a first level of parallelism. These functions could have one or more loops, some of which could be parallelized with OpenMP – a second level of parallelism. Currently, we only permit up to two levels of parallelism for automated hardware synthesis, with Pthreads being the first level and OpenMP being the second.

To evaluate the performance of our parallel accelerators, we used 7 different parallel benchmarks, Black-Scholes, MCML [13], Mandelbrot, Line of Sight, Division, Hash, and Dfsin [14]. Each benchmark was executed in 4 different scenarios: 1) a baseline case where all accelerators operate sequentially; 2) a single level of parallel accelerators using Pthreads; 3) Pthreads combined with loop pipelining, where each Pthread accelerator contains a loop pipelined hardware, and 4) Pthreads combined with OpenMP – nested parallelism. For scenario #4 (Pthreads combined with OpenMP), we experiment with various numbers of Pthreads and OpenMP internal accelerators, for a total of 8 different configurations. The largest configuration has 30 Pthreads with 4 OpenMP internal accelerators, which means that there are essentially a total of 120 accelerators. Note that all 120 accelerators are not necessarily identical, as the 4 OpenMP accelerators only parallelize the loop inside a Pthread function, and there can be other operations done outside the loop. We label architecture configurations as follows: S denotes the sequential baseline case, 4L1 denotes the 4 first-level Pthread accelerators architecture, 4L1-P denotes the 4 first-level Pthread accelerators

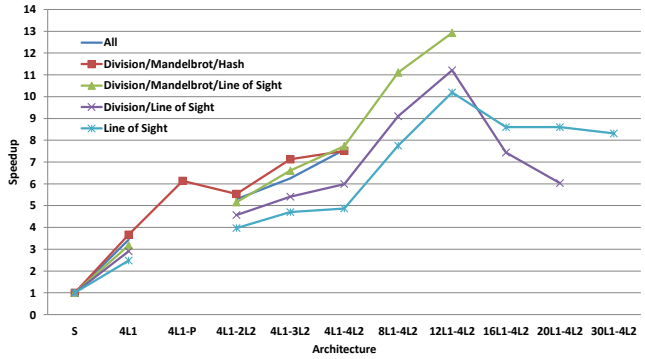


Fig. 4. Geomean speedup ratios.

with loop pipelining, and nL1-mL2 denotes the architecture with n first-level Pthread accelerators with m second-level OpenMP accelerators.

Fig. 4 shows the geometric mean speedup (in wall-clock time) of the different architectures normalized to the baseline case. Since we could not use all the parallelization configurations for all benchmarks, multiple lines are plotted, with each line showing the geomean speedup for a subset of circuits in which the particular configuration could be used[1]. The legend shows which benchmarks are included for each line on the graph. The geometric mean across *all* benchmarks (first line of the legend) shows that the best speedup of 7.6× is observed with the 4L1-4L2 architecture. The 4L1-P configuration is not included in this case, since loop pipelining could not be applied in all benchmarks.

For the benchmarks where loop pipelining could be used (Division/Mandelbrot/Hash), 4L1-P shows 6.17× speedup over baseline, and 4L1-4L2 still shows the best result with 7.51× speedup. However, there are cases where loop pipelining can perform far better. For instance, for the Division benchmark, 4L1-P outperformed all other architectures, even the 20L1-4L2 architecture which has 80 accelerators. This is because a 32-bit division takes 32 cycles in LegUp, using Altera's divider core pipelined to achieve the highest-possible FMax. Since the divider itself is pipelined, it can accept a new input every clock cycle, which is very well suited to loop pipelining. With 4 Pthread accelerators, each of which has only one hardware instance of the loop body, this 4L1-P architecture showed 12.5× speedup over the baseline architecture for the Division benchmark. The biggest speedup in Fig. 4 is 12.9× with the 12L1-4L2 architecture for three benchmarks. Mandelbrot shows the largest single benchmark speedup with 17.2× with the 12L1-4L2 architecture, and 16.6× with the 8L1-4L2 architecture. Overall, as the number of accelerators is increased excessively, the geomean speedups decrease due to reductions in *FMax* and diminishing returns in clock cycle reduction.

B. Loop Pipelining

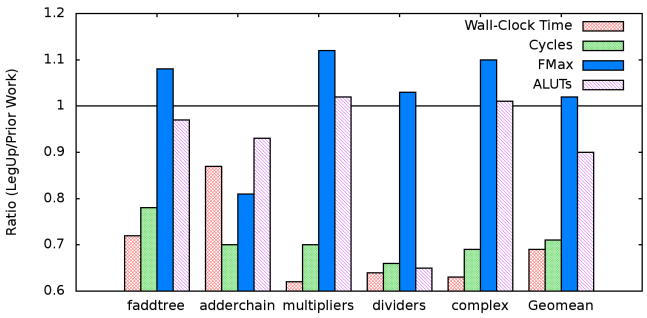
In many `C` applications, the majority of runtime is spent executing critical loops. The high-level synthesis scheduling technique called *loop pipelining*, exploits parallelism across

[1]If we had used a single line, each data point would represent the average for potentially different sets of circuits.

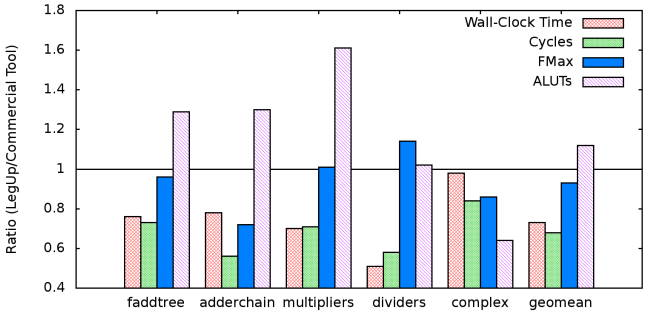
loop iterations to generate hardware pipelines. Loop pipelining increases parallelism and hardware utilization, creating circuits similar to hand-coded hardware architectures. The most common loop pipelining framework used in high-level synthesis is called *modulo scheduling*. Modulo scheduling rearranges the operations from one iteration of the loop into a schedule that can be repeated at a fixed interval without violating any data dependencies or resource constraints. This fixed interval between starting successive iterations of the loop is called the *Initiation Interval* (*II*) of the loop pipeline. The best pipeline performance and hardware utilization is achieved with an *II* of one, meaning that successive iterations of the loop begin every cycle, analogous to a MIPS processor pipeline. Consequently, minimizing the initiation interval can significantly improve pipeline performance.

State-of-the-art HLS scheduling uses a mathematical framework called a system of difference constraints (SDC) to describe constraints related to scheduling [15]. The SDC framework is flexible and allows a wide range of constraints such as data and control dependencies, relative timing constraints for I/O protocols, and clock period constraints. Although loop pipelining has been well studied in HLS, until recently, the SDC approach had not been applied to scheduling loop pipelines due to non-linearities caused by describing the resource constraints in modulo scheduling. Recent work in [16] has extended the SDC framework to handle loop pipelining scheduling by using step-wise legalization to handle resource constraints. This new SDC approach offers compelling advantages over prior methods of modulo scheduling by providing the same mathematical framework for a wide range scheduling constraints. However, the work in [16] uses a greedy approach, where operations are scheduled using a priority function designed to minimize the impact on operations still to be scheduled. Once an operation is scheduled, this decision is final, and if the greedy scheduling is unsuccessful at a given *II*, then [16] increases the *II* and tries again. There are issues applying this greedy approach to more complex loops, particularly the class of loops that contain a combination of recurrences and resource constraints. A greedy modulo scheduling algorithm will not achieve an optimal schedule with the minimum possible initiation schedule if we schedule an operation to a particular time step that later turns out to be wrong. Therefore, greedy scheduling is highly dependent on the chosen priority ordering function.

We have implemented a new backtracking SDC modulo scheduling approach [17] to improve the suboptimal greedy solutions. When the modulo scheduling stops forward progress, we unschedule one of the previously scheduled operations and backtrack, eventually finding the optimal minimum initiation interval. We performed an empirical study on a set of benchmarks containing loop pipelines constrained by resources and limited by recurrences. We compared the new backtracking modulo scheduler to prior work [16] with our results summarized in Fig. 5(a). We also compared our scheduler to a state-of-the-art commercial HLS tool in Fig. 5(b). Our approach achieves a 31% improvement in geomean wall-clock time vs. prior work and a 27% improvement vs. a commercial HLS tool. Our wall-clock time improvement is a result of the lower pipeline initiation interval achieved by our backtracking SDC modulo scheduler when compared to greedy modulo scheduling.



(a) LegUp versus Prior Work [16]



(b) LegUp versus Commercial HLS Tool

Fig. 5. Backtracking SDC modulo scheduling experimental results.

C. Loop Transformations for Loop Pipelining

We also investigated performing loop transformations to further improve loop pipelining performance, specifically for the nested loops. The total cycle latency (L_{total}) of a nested loop with only the innermost loop being pipelined is:

$$L_{total} = [\ II \times (N-1)\ +\ L_{depth}\ ] \times T \qquad (1)$$

where N is the trip count of the pipelined innermost loop, hence the term $II \times (N-1)$ is the cycle latency to start all the inner loop iterations. L_{depth} is the pipeline depth, which is equivalent to the cycle latency of one innermost loop iteration. L_{depth} also corresponds to the cycle overhead to flush out the pipeline in the last iteration. Lastly, T is the number of times the innermost loop executes, which is the product of trip counts of all outer loops. Loop transformations can vary the parameters in the above equation and have a large impact on the overall cycle latency. Currently, we focus on four major transformation techniques. Their utility and application scenarios are explained below, followed by some preliminary results.

Loop interchange is a loop transformation that reorders the execution of loop iterations by swapping an outer loop with an inner loop. Consider a loop nest with depth of 2. Loop interchange can swap the trip counts of the outer loop (T) and the inner loop (N). If T is greater than N, and other variables in the equation are not affected by loop interchange, the total cycle overhead to flush out the pipeline can be reduced by $(T-N) \times (L_{depth} - II)$ cycles.

Loop interchange can also affect *II* by varying the dependence distance carried by the pipelined loop. For instance, both loops in the example code below carry a dependency between iterations; the outer and inner loops carry dependencies with distances of 4 and 1, respectively.

```
for i=4:N
  for j=4:N
    a[i][j] = a[i-4][j] + a[i][j-1];
```

The minimum recurrence *II* is equal to $\lceil \frac{Recurrence\ Length}{Dependence\ Distance} \rceil$, where the *Recurrence Length* is the length (in clock cycles) of a cross-iteration path in the control-dataflow graph, from the node (operation) where a value is computed to the node where it is consumed, and the lengths of individual operations are their minimum latency in clock cycles. Assume the recurrence length to be 4 cycles in the above example. When the inner loop is pipelined, the recurrence *II* is 4 since the inner loop dependence distance is 1. However, if the loops are interchanged, the inner loop dependence distance will become 4 and the recurrence *II* can be reduced to 1. Moreover, the minimum recurrence *II* of 1 cycle can be achieved if the inner loop does not carry a dependency. Meaning that, if the outer loop carries no dependency, but the inner loop does and results in a recurrence *II* greater than 1, loop interchange can be applied to eliminate the dependency carried by the inner loop and hence achieve the minimum recurrence *II*.

Loop skewing is primarily used in conjunction with loop interchange to eliminate an inner loop dependency [18]. Typically, after the transformation, the outer loop trip count is elongated so that the dependency carried by the inner loop can be removed.

Loop distribution (also called loop fission) transforms a loop nest into multiple separated nests. This transformation is useful to create more perfect loop nests and hence more pipeline opportunities. For instance, in the left example below, the statement *s* cannot be pipelined since it is not inside the innermost loop. With loop distribution, the original loop nest is split into two perfect loop nests where all operations can be pipelined.

```
for i=1:N {
s: b[i][j] = i+j;
   for j=1:N
      a[i][j] = i*j;
}
```

⇨

```
for i=1:N
s: b[i][j] = i+j;

for i=1:N
   for j=1:N
      a[i][j] = i*j;
```

Loop fusion performs the inverse transformation of loop distribution. This transformation is particularly useful in combining perfect loop nests that have the exact same loop bounds. With fewer independent loop nests and more instructions in the merged loop, the hardware throughput can be improved as more instruction-level parallelism is achieved.

TABLE III
TRANSFORMATIONS APPLIED AND SPEED-UP ACHIEVED.

Benchmark	Transformations Applied	Cycle Latency [$\times 10^3$]		Speed-up
		Original	Optimized	
reg_detect	distribution	2,710	1,166	2.32×
gemver	interchange, fusion	3,007	1,257	2.39×
adi	interchange	10,432	4,135	2.52×
seidel-2d	skew, interchange	7,179	831	8.64×

To demonstrate the potential of improving loop pipeline performance using these transformations, we manually apply the transformations on four kernels that are selected from the PolyBench/C 3.2 benchmark suite [19], and we evaluate the cycle latency through ModelSim simulation. In the experiment, all of the innermost loops are always pipelined. Table III lists the transformations applied in each kernel and compares the cycle latency before and after the transformations. The cycle latency speed-up achieved in each kernel ranges from 2.32× to 8.64×. These loop transformations do not require extra resource usage and they do not reduce the circuit clock frequency. Our next step will be to develop a method for determining the best transformations to apply based on the input program and then automate the optimization flow as part of the LegUp synthesis flow.

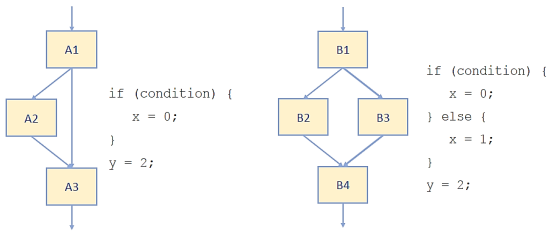


Fig. 6. Pattern A *(left)* and Pattern B *(right)*.

D. Control-Flow Graph Modifications

We also implemented an approach to improve the speed of HLS-produced hardware by transformations in the control-flow graph (CFG) of the program. The basic idea is to identify patterns within the CFG and selectively merge the basic blocks associated with these patterns into a single "mega" basic block, similar to hyperblocks using if-conversion [20]. Currently, we look for the patterns of basic blocks shown in Fig. 6, which we refer to as Pattern A and Pattern B. By collapsing such patterns into mega blocks, instructions that were originally scheduled after branch(s) may be able to execute earlier and in parallel with other instructions, thereby increasing instruction-level parallelism and improving performance. For example, with Pattern A, instructions in block A2 may be able to execute in parallel with instructions in block A1 after collapsing. We apply the collapsing judiciously by first profiling the application in software to determine the number of times each block is executed under a typical input dataset, then scheduling the program with and without the collapsing, and ultimately, determining conclusively whether or not collapsing a given pattern instance is profitable to cycle latency. Note that we expect such collapsing to be detrimental to power consumption, as computations within intermediate basic blocks will *always* execute, instead of executing only for a certain branch outcome.

Regarding the implementation, to ensure functional correctness, we reject patterns for which the intermediate blocks, i.e. block A2 of Pattern A, and blocks B2, B3 of Pattern B, contain `call` instructions, as function calls may result in undesired modifications of memory. However, store instructions are permitted and are handled by predication. This is achieved by selecting to store to either a valid or an invalid `null` address based on the branch condition (which is nevertheless computed

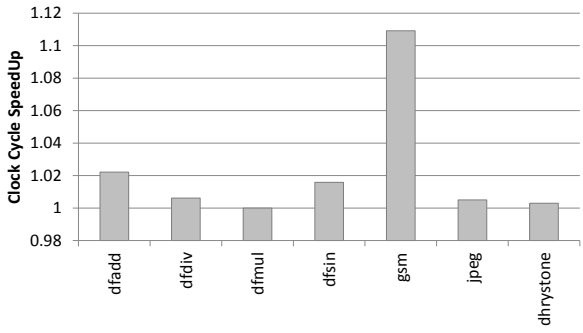


Fig. 7. Cycle latency speedup for benchmarks exhibiting valid merge patterns.

in the mega block, but not used to control a branch). `phi` instructions in tail blocks A3 and B4 are replaced by `select` instructions in the mega block.

To evaluate the performance of the modified hardware we used the CHStone suite of HLS benchmarks, as well as 4 other local LegUp test cases. Fig. 7 plots the clock cycle speedup for benchmarks which contained valid merge patterns, i.e. benchmarks which contained at least one merged basic block pattern in the final IR used for Verilog generation. For the `gsm` benchmark, over 10% improvement is observed, with more modest improvements seen for the other circuits. Future work will broaden the approach to larger and more complex patterns of basic blocks based on their dominance relationships.

V. Raising HLS Hardware Quality

In this section, we highlight work underway to improve the quality of hardware produced by HLS.

A. Memory Synthesis

Memory presents challenges in high-level synthesis when using `C` as the user specification language. `C` targets a von-Neumann-style computing architecture, which can lead to memory bottlenecks in highly parallel algorithms. However, in HLS we can generate a custom memory architecture for our particular target application depending on the memory locality and program access patterns. In many cases, we cannot determine the exact memory locations that each `C` pointer could access at compile time. This is called the pointer aliasing problem. Most commercial HLS tools avoid handling pointer aliasing and instead limit the program input to simple arrays whose accesses can be determined statically. LegUp does not have this limitation on program input. Therefore, for memories where the accesses cannot be resolved, we must allocate them into a global memory, where we can dynamically resolve each pointer address to a place in memory at run-time.

The initial version of LegUp allocated all arrays specified in the `C` program into a global shared memory. Each `C` array is stored in a separate FPGA on-chip block RAM, with a width that matches the data stored in the array. Each BRAM is identified by a 9-bit tag. All memory accesses in the circuit are fed into a shared memory controller, which steers the access to the correct RAM depending on the tag bits. There are a few limitations to this approach. First, the memory controller output contains a wide multiplexer, which grows linearly with the number of memory allocations in the `C` code. This multiplexer affects FMax, and forced LegUp to use a memory latency of two cycles. The second issue relates to the program call stack. Each `C` function corresponds to a Verilog module in the final LegUp-synthesized hardware. Functions lower in the call stack of the program are instantiated deeper in the module hierarchy of the circuit. At each level of hierarchy, we create a multiplexer to steer the memory signals, which can affect FMax for programs with a deep call graph.

We mitigate these problems by using *local memories*. We partition the memory space into global memory, which allows us to dynamically resolve pointer aliasing at runtime, and local memory, when we can statically determine the memory being accessed. In the case of the hybrid flow we have a third memory form: memory allocated by the processor and accessed through the processor cache. We partition all arrays into either local or global memory. Local memory is used for `C` arrays that are only used in one function. In this case, we instantiate a local BRAM within the module corresponding to that `C` function. This also works for all constant arrays – which can be duplicated inside all functions that access the constant.

Local memory can shrink the number of multiplexers in the design by avoiding accesses to the shared memory controller. Local memories can also be accessed in parallel without being limited by the two ports on the shared memory controller. We do not support local memory in some cases. If the user allocates an array and then passes a pointer to that array to another function, we mark the array as global memory. Another case is pointer indirection, where the pointer to an array is stored inside another array. In this case, we also mark the array which has been referenced as a global memory. We have found that the shared memory controller is typically on the critical path, so local memories improve FMax.

Another problem with LegUp's existing approach is poor utilization of Stratix IV M9K memory blocks in the shared memory controller. We found that an Altera synchronous on-chip RAM with only a few words will be synthesized to use an entire M9K block on the FPGA, or effectively 9Kb of memory. To mitigate this issue, we group arrays by word width and instantiate them into one large block RAM. We found a significant M9K savings by packing many memories into the same M9K using this grouped memories approach. Furthermore, using local RAMs in isolation also improves M9K usage because Quartus is able to optimize the smaller RAMs away (implementing them in LUT RAM).

Grouped RAMs requires an offset for each array to find the array's location in the larger block RAM. An address now consists of: Tag + Offset + Index. However, this offset can cause additional addition operations in the final hardware compared to the previous Tag + Index approach. We handled this by padding the block RAM and making the offset divisible by the number of words in each array. By doing this, we can calculate the address using simple concatenation of Tag, Offset, and Index. This saves area (fewer adders) and improves FMax at the cost of memory bits. However, these memory bits are typically wasted anyway by under-utilized M9Ks in the original LegUp implementation.

We studied the impact of memory partitioning using the CHStone benchmarks [14]. The CHStone benchmarks include

golden input and output test vectors, allowing us to synthesize the circuits with a built-in self-test. We used these test vectors to simulate the circuits in ModelSim and verify correctness. We targeted the Stratix IV [21] FPGA (EP4SGX530KH40C2) on Altera's DE4 board [22] using Quartus II 11.1SP2 to obtain area and FMax metrics. Quartus timing constraints were configured to optimize for the highest achievable clock frequency. We considered four scenarios for comparison: 1) the default LegUp flow (Orig); 2) grouping RAMs in the global memory controller (Group); 3) partitioning memory into local RAMs and global RAMs (Local); and, 4) combining both RAM grouping in the global memory controller and memory partitioning (Both).

Table IV gives speed performance results for these four scenarios. The "Cycles" column is the total number of cycles required to complete the benchmark. The "FMax" column provides the $FMax$ of the circuit given by the Quartus. The "Time" column gives the circuit wall-clock time: $Cycles \cdot (1/FMax)$. Ratios in the table compare the geometric mean (geomean) of the column to the respective geomean in the default LegUp flow. The geomean FMax was improved by 11% by combining local RAMs with grouping, due to a smaller multiplexer required in the shared global memory controller. Partitioning memory into local RAMs hardly improved the geomean cycles indicating that the schedule of these benchmarks was not constrained by the global shared memory ports. Overall geomean wall-clock time performance was improved by 10% by combining local RAMs with grouping RAMs in the memory controller, with a portion of the overall improvement coming from the local RAMs and a portion from the grouping.

Detailed area results could not be included for space limitations; however, we found a significant improvement in the effective memory bits required during synthesis, which decreased by 29% by grouping RAMs. When we combined local RAMs with grouping, we found that the reduction was 67%. By combining local RAMs with grouping we found geomean ALUTs decreased by 18% and geomean registers decreased by 12%. This was due to less multiplexing in the shared memory controller and also registers being packed into the local block RAMs.

B. Multi-Cycling

Significant clock frequency speedups have also been achieved in the LegUp-generated datapath through the use of multi-cycle paths. A multi-cycle path is a register-to-register path of combinational logic which is allowed multiple clock cycles to "complete" (i.e. a transition along the path is permitted to propagate for more than one clock cycle). Such paths occur frequently in LegUp circuits when an instruction completes more than one cycle before the result of its computation is consumed – we refer to such a path as having *multi-cycle slack*. A naturally occurring multi-cycle path is illustrated in Fig. 8, in which the multiplication E*F, not required until clock cycle 3, is allowed two cycles to complete. Analysis of the LegUp schedule is automatically performed to identify all instances of multi-cycle slack and print constraints for Altera's Quartus II Timing Analyzer so the timing analyzer "knows" the path is permitted extra cycles.

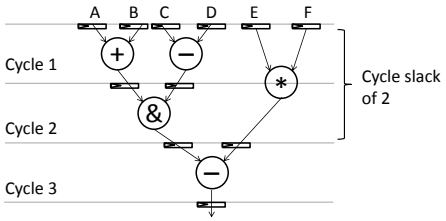


Fig. 8. Illustration of a multi-cycle path.

We calculate these cycle delays by traversing all register-to-register paths post-scheduling, including paths scheduled across basic block boundaries.

In addition to this static timing analysis to discover multi-cycle paths, LegUp also performs datapath de-pipelining to create additional multi-cycle opportunities. In general, hardware generated by LegUp is pipelined and able to process new data each clock cycle, including loops, dividers and floating point operations. However, the level of instruction-level parallelism inferred from the `C` source does not always permit an initiation interval of one for these pipelines. While circuits can accept new data each clock cycle, this is rarely observed in `C` benchmarks, which describe algorithms sequentially and are more often translated into hardware where only one or a few hardware units are active at a time. Because such datapaths do not always benefit from pipeline parallelism, an alternative is to fully de-pipeline datapaths and instead, designate them as multi-cycle paths (of equivalent latency).

Multi-cycle paths have many benefits compared to fully pipelined paths. Eliminating registers both saves area and removes a portion of register-to-register delay: clock-to-Q delay, setup time, clock skew. De-pipelining also allows synthesis tools to optimize logic across register boundaries. Note also that registers in FPGAs reside in specific predefined locations on the die and there is consequently a delay "cost" (albeit small) associated with physically routing a path to a register location – multi-cycle paths do not carry this restriction. Moreover, de-pipelining datapaths allows the delays of computations along the path to be averaged such that no one computation is critical.

Multi-cycle constraints target solely datapaths, so not all circuits show speedups (some circuits may have critical paths that do not reside in the datapath). Nevertheless, while multi-cycle work is ongoing in LegUp, 4 of the 12 CHStone benchmarks already show a reduction in total execution time of 10% or greater, with `dfmul` improving by 16% and `dfdiv` by 25%. The greatest speedups are due to cycle slack analysis across basic blocks, in which paths are identified that may complete in more cycles than Quartus was able to infer.

A number of additional optimizations are also under development. For example, multi-cycle paths currently slow down the clock frequency of `gsm` by 3×. In this circuit, numerous multi-cycle paths share the same source and destination registers, but pass through different basic blocks and therefore have unbalanced latencies. LegUp prints a single multi-cycle slack constraint for all register-to-register pairs, and in the case of multiple paths between two registers, LegUp must select the shortest latency of all paths to prevent timing violations. This makes the high-latency paths between these

TABLE IV
MEMORY ARCHITECTURE PERFORMANCE RESULTS

	Cycles				FMax (MHz)				Time (μs)			
Benchmark	**Orig**	**Group**	**Local**	**Both**	**Orig**	**Group**	**Local**	**Both**	**Orig**	**Group**	**Local**	**Both**
chstone/adpcm	27026	27026	26104	26104	159	135	132	153	170.0	200.2	197.8	170.6
chstone/aes	9452	9452	9372	9372	125	139	139	156	75.6	68.0	67.4	60.1
chstone/blowfish	186428	186428	186428	186428	163	175	180	189	1143.7	1065.3	1035.7	986.4
chstone/dfadd	746	746	746	746	240	234	249	249	3.1	3.2	3.0	3.0
chstone/dfdiv	1956	1956	1956	1956	232	241	215	215	8.4	8.1	9.1	9.1
chstone/dfmul	272	272	272	272	255	246	254	254	1.1	1.1	1.1	1.1
chstone/dfsin	59132	59132	59132	59132	162	154	147	147	365.0	384.0	402.3	402.3
chstone/gsm	5908	5908	5906	5906	175	181	204	204	33.8	32.6	29.0	29.0
chstone/jpeg	1252416	1252416	1232666	1232666	101	106	101	114	12400.2	11815.2	12204.6	10812.9
chstone/mips	6228	6228	6228	6228	231	251	264	264	27.0	24.8	23.6	23.6
chstone/motion	8420	8420	8420	8420	233	229	224	249	36.1	36.8	37.6	33.8
chstone/sha	258042	258042	256500	256500	192	197	221	246	1344.0	1309.9	1160.6	1042.7
dhrystone	7760	7760	7760	7760	173	220	218	266	44.9	35.3	35.6	29.2
Geomean	**14097**	**14097**	**14026**	**14026**	**181.7**	**186.7**	**189**	**201**	**78**	**75**	**74**	**70**
Ratio	**1**	**1**	**0.99**	**0.99**	**1**	**1.03**	**1.04**	**1.11**	**1**	**0.97**	**0.96**	**0.90**

two registers critical in the overall circuit. To solve this, LegUp sets `-through` constraints on these paths, specifying different multi-cycle slacks depending on intermediate signals in a path. A challenge however is that synthesis tools may optimize away these intermediate signals, and while directives can be added to preserve intermediate wires, this introduces additional delays and increases circuit area. One current enhancement therefore is selecting a *minimal* set of intermediate signals to preserve, while still ensuring that each path can be uniquely constrained, thereby trading some additional delay for more precise slack allocation.

VI. INSPECT: LEGUP'S DEBUGGING PLATFORM

If HLS is to be useable by software engineers, new debugging and visualization methodologies are needed. LegUp's debugging framework, called *Inspect* [23], aims to provide a `gdb`-like experience for HLS hardware, offering features such as `C` code stepping, break points, and variable watching.

Through a GUI, the user is able to step through the `C` code and inspect the values of variables. However, unlike a software debugger which executes the program on a processor, in this case, the run-time program data is extracted from a hardware execution of the circuit. There are two modes of operation: 1) simulation mode, where signal values are extracted from a ModelSim simulation; and 2) silicon debug mode, where signal values are extracted from an actual HW execution on an FPGA by using Altera's SignalTap II logic analyzer [24]. Simulation mode is useful for debugging small and medium sized applications and it provides full visibility into the design as all signals can be examined. The silicon debug mode is useful for analyzing timing-related bugs, transient runtime errors (SEUs), bugs in the interface and so on that are "invisible" at the RTL level. However, not all variables (hardware signals) can be inspected in this mode because of pin, memory and interconnect limitations on the FPGA.

To relate the software to the hardware, Inspect uses a debug database, which we create automatically during HLS. The database, implemented in MySQL [25], contains the links between the program, its internal representation within LLVM, and finally, the hardware. For a given `C` variable, the database allows us to find the corresponding HW signal. Likewise, for a given `C` statement, we can find the FSM states where the statement's computations occur.

Fig. 9 shows the Inspect GUI. At the top left, buttons allow the user to step and run/continue the program/HW. There are two styles of stepped execution available: *statement* step and *cycle* step. Statement step advances the execution by one `C` statement, which may take multiple clock cycles in the hardware. Cycle step advances the execution by a single clock cycle. Observe the left-most panel of the GUI containing the line numbers. The user may click on a line number to set a breakpoint at the corresponding line. Continuing right, the next panel shows the `C` code, with the currently executing line highlighted. The next panel shows the corresponding LLVM IR. The panel below shows the corresponding Verilog code being executed, in this case, a *compare* operation arising from the inner-most loop of the `C`. The right-most panels show the HW signals active in the current scope, and, at the bottom, the variable values, as extracted from the HW. At the top-right of the GUI, the current FSM state is displayed.

VII. COMPARISON AGAINST LEGUP 3.0

We compared the quality of results provided by the current LegUp tool, with that provided by the 3.0 release (early 2013). We use the CHStone HLS benchmark suite [14], and the dhrystone integer benchmark, and we target the Altera Stratix IV FPGA. A difficult-to-meet timing constraint was applied for all runs. Note that while the CHStone benchmarks are commonly used in HLS research, they generally do not have opportunities for loop pipelining, and as such, the loop optimizations described in previous sections were not applied.

Table V shows the results. Column 1 gives the name of each benchmark. The next four columns give the wall-clock time, cycles, FMax (in MHz), and the number of Stratix IV ALMs for LegUp 3.0, respectively. The right-most column of the table gives the the ALMs for the current LegUp. The third last row presents geometric mean data for LegUp 3.0; the second last row provides geometric mean data for the current LegUp; the last row presents the ratio of the geometric mean vs. LegUp 3.0. On all metrics, we see significant quality-of-results improvements vs. the prior release. On average, wall-clock time is improved by 27%, cycle count by 13%, FMax

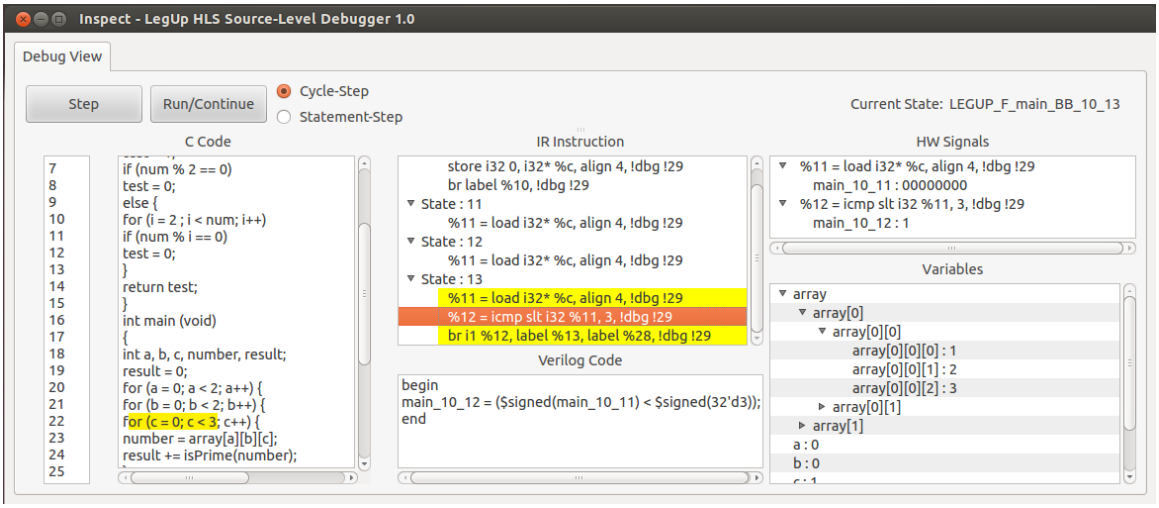


Fig. 9. GUI for Inspect debugger.

TABLE V
QUALITY-OF-RESULTS COMPARISON WITH LEGUP 3.0.

	LegUp 3.0				Current
Benchmark	**Time**	**Cycles**	**FMax**	**ALMs**	**ALMs**
adpcm	221	24106	109	6951	5201
aes	69	9891	144	7701	7353
blowfish	1066	179561	168	4966	4228
dfadd	3	775	235	3240	1910
dfdiv	8	1970	235	6138	3967
dfmul	22	3067	141	8588	1186
dfsin	471	59765	127	11470	11355
gsm	36	5367	150	4886	4097
jpeg	13077	1237646	95	17633	13799
mips	23	5475	236	1499	1201
motion	27	6363	237	1906	1772
sha	1383	257532	186	6650	4898
dhrystone	37	6813	186	2986	2261
Geomean (3.0)	**96.2**	**16095**	**165.9**	**5291**	
Geomean(curr)[2]	**69.8**	**14026**	**201.5**	**3671**	
Ratio:	**0.73**	**0.87**	**1.21**	**0.69**	

by 21%, and area by 31%.

VIII. CONCLUSIONS

LegUp is an open-source high-level synthesis tool being developed at the University of Toronto that allows a C program to be automatically converted to hardware, or alternately, to a hybrid system with a processor and accelerators. The tool remains under active development, with current thrusts being: 1) adding support for ARM processors, 2) increasing hardware parallelism, 3) hardware quality improvements, and 4) visualization and debugging. Visit `http://legup.eecg.toronto.edu` to learn more and download LegUp.

REFERENCES

[1] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 2, 2013.

[2]See "Both" columns of Table IV for Time, Cycles, and FMax

[2] J. Cong and Y. Zou, "FPGA-based hardware acceleration of lithographic aerial image simulation," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 2, no. 3, Sep. 2009.
[3] P. G. Paulin and J. P. Knight, "Force-directed scheduling in automatic data path synthesis," in *IEEE/ACM DAC*, 1987, pp. 195–202.
[4] *Altera SDK for OpenCL*, Altera, Corp., 2014.
[5] *Vivado High-Level Synthesis*, Xilinx, Inc., 2014.
[6] *Cyclone V SoC hard processor system*, Altera, Corp., 2014.
[7] *LLVM Compiler Project (http://www.llvm.org)*, 2010.
[8] *The Tiger "MIPS" processor.*, University of Cambridge, http://www.cl.cam.ac.uk/teaching/0910/ECAD+Arch/mips.html, 2010.
[9] 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*, 2011, pp. 61–68.
[10] Altera, "Processor selector," *http://www.altera.com/devices/processor/selector/proc-processor-selector.jsp*, 2014.
[11] *Altera DE1-SoC Board*, Altera, Corp., 2014.
[12] J. Choi, J. Anderson, and S. Brown, "From software threads to parallel hardware in FPGA high-level synthesis," in *IEEE International Conference on Field-Programmable Technology*, 2013, pp. 270–279.
[13] *Monte Carlo Simulations.*, Oregon Medical Laser Center, http://omlc.ogi.edu/, 2007.
[14] 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," *Jour. of Information Processing*, vol. 17, pp. 242 – 254, 2009.
[15] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on SDC formulation," in *IEEE/ACM DAC*, 2006, pp. 433–438.
[16] Z. Zhang and B. Liu, "SDC-based modulo scheduling for pipeline synthesis," in *ICCAD*, San Jose, CA, 2013.
[17] A. Canis, S. Brown, and J. Anderson, "Modulo SDC scheduling with recurrence minimization in high-level synthesis," in *Int'l Conf. on FPL*, 2014.
[18] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Comput. Surv.*, vol. 26, no. 4, pp. 345–420, Dec. 1994.
[19] L.-N. Pouchet and U. Bondugula, *PolyBench/C - the Polyhedral Benchmark Suite*, December 2012.
[20] S. A. Mahlke, D. C. Lin, and e. Chen, "Effective compiler support for predicated execution using the hyperblock," in *ACM SIGMICRO*, vol. 23, no. 1-2. IEEE Computer Society Press, 1992, pp. 45–54.
[21] *Stratix-IV Data Sheet*, Altera, Corp., 2010.
[22] *DE4 Dev. and Education Board*, Altera, Corp., 2010.
[23] N. Calagar, J. Anderson, and S. Brown, "Source-level debugging for FPGA high-level synthesis," in *Int'l Conf. on FPL*, 2014.
[24] *SignalTap II Logic Analyzer User Guide*, Altera, Corp., San Jose, CA, 2013.
[25] *MySQL Open Source Database*, http://www.mysql.com, 2014.