From Pthreads to Multicore Hardware Systems in LegUp High-Level Synthesis for FPGAs

Jongsok Choi, *Student Member, IEEE*, Stephen D. Brown, *Member, IEEE*, and Jason H. Anderson, *Member, IEEE*

***Abstract*— In the last decade, processor speeds have remained fairly stagnant, and to improve performance further, the industry started to increase the number of processor cores. The use of specialized hardware, such as field-programmable gate arrays (FPGAs), has also been on the rise. The traditional design methodology for FPGAs, however, requires hardware knowledge, which makes the platform inaccessible to software engineers. High-level synthesis (HLS) tools aim to resolve this issue by allowing software design methodologies to be used for FPGAs. However, HLS remains difficult to use for many software engineers, as there are tasks, such as system integration, which is still mostly a manual process. Consequently, creating a multicore hardware system on an FPGA is not feasible for most software engineers. To this end, we provide an HLS framework, which can automatically generate a multicore hardware system from software. We provide support for POSIX threads, which can be compiled to concurrently executing hardware cores that can be used in a processor–accelerator hybrid system, or in a hardware-only system without a processor. With this, we show that we can create multicore FPGA systems that can provide significant benefits in performance and energy-efficiency compared with hardware executing sequentially, and software executing on MIPS/ARM/`x86` processors.**

***Index Terms*— Field-programmable gate array (FPGA), high-level synthesis (HLS), parallel hardware, POSIX threads (Pthreads), system-on-chip (SoC).**

I. Introduction

HIGH-LEVEL synthesis (HLS) raises the design abstraction of hardware to software, allowing a user to automatically generate a circuit description from a high-level software specification. Its promise to make hardware design easier and reduce the time-to-market is being acknowledged by many engineers, evidenced by the widespread adoption of the technology and increasing research in both academia and industry, including [1]–[6]. A number of academic groups have built their own HLS tools [7]–[9], and a variety of commercial HLS offerings are also available [10]–[15], including those which have been used to design chips in production [16]. While some of these HLS tools may differ in terms of their input languages or the set of features they provide, their overarching goal largely remains the same—making hardware design easier for hardware engineers, and allowing hardware to be designed by software engineers.

With the end of clock frequency scaling in the last decade, modern processors have sought to improve performance by increasing the number of cores. While a state-of-the-art consumer CPU may have 10+ cores, the Intel Xeon Phi, used in supercomputers, has more than 60 cores [17]. The Nvidia K80 GPU has almost 5000 CUDA cores [18]. All of these CPU/GPU cores are generally utilized by parallel software programming methodologies, such as POSIX threads (Pthreads) [19], OpenMP [20], OpenCL [21], and CUDA [22], where the programmer explicitly specifies the parallelism in the code.

FPGAs have also been growing in size. The Xilinx UltraScale XCVU440 FPGA has 4.4 million logic cells and more than 20 billion transistors on a single chip [23]. The large number of logic cells can be tailored to create a custom multicore hardware system. Creating a core in hardware involves either writing RTL code, importing a core from an IP library, or using an HLS tool to generate a core from software. To create a multicore system, one can manually stitch the cores together using RTL, or use a system integration tool, such as Qsys [24], or Vivado IP Integrator [25]. This integration process can be cumbersome for a hardware engineer, but may not even be possible for a software engineer. In addition, bringing up off-chip interfaces, such as DDR3, is an arduous task even for hardware engineers.

In this paper, we present a methodology where a custom multicore hardware system can be automatically generated from a standard software program. In particular, we use Pthreads, a well-known parallel programming methodology, to generate parallel cores from software threads. In addition to the compute cores, the interconnect and memories are also generated to create a complete system-on-chip (SoC).

Prior work in [26] showed that we can accelerate a multithreaded software program by synthesizing software threads to parallel hardware accelerators, with the remainder (nonthreaded portion) of the program executing on a soft MIPS processor. The prior work was limited to a single type of processor architecture (MIPS), with a fixed memory architecture (on-chip caches backed by off-chip memory), on a single FPGA board (the Altera DE4 board). In recent years, FPGA vendors have introduced SoC chips with a hard ARM processor integrated on the same die as the FPGA [27], [28]. The ARM processor, typically running at speeds between 800 MHz and 1.5 GHz, runs much faster than the FPGA fabric, and can execute software programs significantly faster

Manuscript received August 30, 2016; revised January 15, 2017 and May 2, 2017; accepted June 4, 2017. *(Corresponding author: Jongsok Choi.)*

The authors are with the Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON M5S 3G8 Canada (e-mail: jongsok.choi@mail.utoronto.ca; brown@eecg.toronto.edu; janders@eecg.toronto.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2017.2720623

than what was previously possible with soft processors, such as with the MicroBlaze [29] or the NIOS II [30].

In this paper, we provide HLS support for using the ARM hard processor system (HPS) on the Altera Arria V SoC FPGA, to realize an ARM processor–accelerator hybrid system, where Pthread functions are accelerated in hardware, and the remaining software segments are executed in software on the hard ARM processor. We compare this with the previously available method of using the soft MIPS processor. Bringing up the ARM processor, especially in bare metal mode, is an onerous and troublesome task for both software and hardware engineers. With this paper, we provide an SoC system that has been completely set up, so that the ARM processor can simply be programmed and used through `make` targets. The user can also choose to run the ARM processor with a Linux OS as opposed to bare metal. To improve memory bandwidth, we also provide direct memory access (DMA) support, so that large amounts of data can be transferred in bursts between off-chip DDR3 memory and on-FPGA hardware accelerators.

For some applications, it may be beneficial to compile the entire program to purely hardware, instead of using a processor–accelerator hybrid system. ARM SoC FPGAs are still relatively nascent, with only a handful of SoC FPGAs on the market, and soft processors can add significant area/power overheads. It is therefore desirable if Pthreads can also be used in HLS without a processor in the system. To this end, we also describe our support for the hardware-only flow, where a multithreaded software program can be compiled entirely to hardware, with parallel threaded modules executing concurrently within a hardware system. By removing the processor requirement, we believe that our HLS support for Pthreads can be applied to a wider range of applications.

We use the LegUp HLS framework [7] from the University of Toronto in this paper. The contributions are as follows.

1) HLS support for Pthreads to automatically generate a parallel hardware-only system or an ARM processor/parallel-accelerator hybrid system.
2) A board support package for the Altera Arria V SoC Development Board, where its components, such as the ARM HPS and DDR3 memory, are configured and set up for easy use. We also provide a complete toolchain for running the ARM processor in bare metal or with a Linux OS.
3) Providing software and hardware support for DMA data transfers. We provide a software DMA library that can be used to control DMA operations from software, and a hardware double buffering module, which allows an accelerator to continuously execute while its data are moved in and out of the accelerator.
4) An analysis of the performance, area, and energy consumption of hardware-only and processor–accelerator hybrid systems. We demonstrate that our ARM hybrid system can show significant speedup and energy-efficiency benefits compared with software executing on the ARM, as well as on two different `x86` processors, the Intel i7-4770K CPU and the Intel Xeon E5-1650 CPU.

The rest of this paper is organized as follows. Section II presents related work. Section III provides a short overview of Pthreads in software. Sections IV and V discuss how we implement Pthreads in hardware and describe the hardware architectures of hardware-only and hybrid systems. An experimental study is described in Section VI and the conclusions with recommendations for future work are presented in Section VII.

II. Related Work

A number of previous works have implemented hardware support for threads on FPGAs. The work in [31] describes a framework, which uses Pthreads to generate hardware accelerators at runtime by running an FPGA CAD tool on an on-FPGA ARM processor with an OS. However, the FPGA synthesis tool could not actually run on the embedded ARM processor, and their results are based on their own C++ simulator. In this paper, we use both cycle-accurate ModelSim simulation and on-board execution to obtain accurate runtimes. We also provide Pthreads support for the hardware-only flow, which does not require a processor in the system. HybridThreads (hthreads) provide a library allowing thread execution on a hybrid CPU/FPGA system [32]. While similar to Pthreads, hthreads is not a standard software library, and thus not portable to other platforms.

On the commercial front, Xilinx's Vivado HLS [10] is related in terms of its ability to compile software to a hardware core. However, Vivado HLS does not support Pthreads, and thus, in order to create multiple concurrent cores, the user needs to instantiate them manually. Xilinx's SDSoC [35], which targets software engineers, can generate an SoC system automatically, but does not support using a standard parallel programming methodology to create parallel hardware cores. The tool uses a vendor-specific pragma, `#pragma SDS async()` [36], to create parallel hardware cores, which while useful, is a nonstandard methodology and is not portable to other platforms. The use of the vendor-specific pragma also implies that hardware behavior is different from software behavior, as the pragma specifies parallel hardware, but when the software is compiled with a standard compiler, such as `gcc`, it executes sequentially. This is not the case with Pthreads in this paper, where both software and hardware execute in parallel. It is also worth mentioning Xilinx's SDAccel [37] and Altera's OpenCL SDK [11], both of which support compiling OpenCL kernels to pipelined hardware. OpenCL is typically used for massively parallel applications, traditionally on GPUs and now on some FPGAs. The programming model is different for OpenCL, where there is a clear division between the host code, which executes in software on an `x86`/ARM processor, and the kernel code, which is compiled to hardware. The user explicitly needs to make the separation, with the host code written in C/C++, and the kernel code written in OpenCL—there is no automatic software/hardware partitioning involved. In this paper, we automatically partition the program, so that Pthread functions are accelerated using hardware, with the remaining program executed on an MIPS/ARM processor. No user intervention is required in this process.

As previously mentioned, we have made many improvements from the prior work in [26], which implemented the initial support for Pthreads for the soft MIPS processor–accelerator system. In this paper, we have implemented support for the hard ARM processor on the Arria V SoC FPGA, where the processor can either run bare metal or a Linux OS. We have created a convenient flow that automatically synthesizes Pthreads-based parallel software Pthreads to an ARM/FPGA SoC. We have also significantly improved the memory architecture, by implementing support for a points-to analysis, a static code analysis technique that establishes which variable/array a pointer can point to. With this, we boost memory parallelism through more simultaneous accesses (described in Section V-A). We have also added DMA support with double buffering for high-memory-bandwidth data transfers between accelerators and off-chip memory. Finally, we have implemented the hardware-only flow with Pthreads, which allows multiple hardware cores to execute concurrently without requiring a processor in the system. In both the hardware-only flow and the hybrid flow, a hardware core can optionally be shared across multiple threads (described in Section IV-A).

III. Pthreads in Software

This section describes briefly how Pthreads can be used to express parallelism in software. Consider the following code snippet.

```
// fork threads
for (i=0; i<N; i++) {
  pthread_create(&threads[i], NULL,
                 vector_add, &data[i]);
}

// join threads
for (i=0; i<N; i++) {
  pthread_join(threads[i], NULL);
}
```

The example forks `N` threads with `pthread_create`, each of which executes the `vector_add` function, with an element of the `data` array given as its argument. The `N` threads are joined with `pthread_join`, which waits for the thread specified by its thread variable (`threads[i]` in this case) to terminate. The return value can be retrieved via the second argument of `pthread_join` (`NULL` in this case).

Reference [26] shows the Pthread APIs that we support in compiling to hardware. We have selected those functions that we believe are to be the most widely used aspects of Pthreads in software, and consequently the most beneficial to provide support for automatic synthesis to hardware.

IV. Pthreads to Parallel Hardware-Only System

In this section, we describe how Pthreads are compiled to a hardware-only system (without a processor), which comprises multiple hardware modules concurrently executing within a larger hardware system. Fig. 1 shows the hardware-only flow in LegUp HLS. It comprises three major steps, frontend, intermediate representation (IR) transformations, and hardware backend. The frontend step invokes `Clang`, LLVM's frontend compiler, which receives a C program as input and

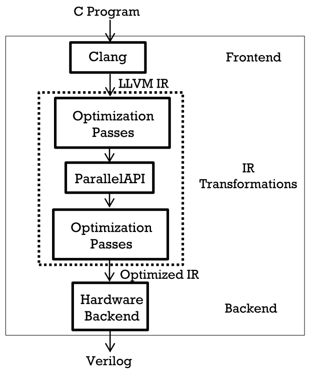

Fig. 1. Hardware-only flow in LegUp HLS.

produces LLVM IR, which is target-independent assembly-like code. After converting C code to LLVM IR, all HLS operations are performed on the IR. In the IR transformations step, the program that goes through a series of compiler optimization passes to optimize the program. These passes include standard optimization passes included in LLVM (such as dead code elimination or constant propagation), as well as custom passes that we have written ourselves to optimize for the output hardware. One of custom passes related to this paper is the `ParallelAPI` pass, which is responsible for handling Pthread functions. After the `ParallelAPI` pass, more optimization passes are applied to the IR, and finally, the optimized IR is inputted to the hardware backend step, which executes the HLS steps of allocation, scheduling, and binding, to generate hardware specified in Verilog. The backend is also responsible for creating concurrent hardware modules and creating the arbitration logic needed to share memories. In LegUp, by default, each thread is compiled into an independent hardware module (e.g., the example shown previously gets compiled to `N` instances of the `vector_add` module in hardware).

In software, using Pthreads requires the `POSIX` library and an OS to manage threads. We have neither the POSIX library nor the OS in hardware, hence the research challenge here is how we can manage hardware threads without an OS or a threading library, and with minimal performance/area overhead. To this end, the `ParallelAPI` pass is responsible for: 1) removing the dependence on the Pthread library and 2) creating logic to manage threads. For 1), the pass replaces the calls to Pthread API functions, with direct calls to the threaded functions (functions executed on threads). For instance, in the above-mentioned Pthread example, `pthread_create` is replaced with a direct call to `vector_add`. `Pthread_join` uses the thread variable to determine which thread to join. At compile time, `pthread_join` does not "know" which function it intends to join. This is determined at runtime based on the thread variable passed in as its argument. Hence, we needed a methodology where we can keep track, at runtime, of which thread is accelerating which function, as well as which thread is running on which *instance* of hardware for that function. The `ParallelAPI` pass replaces all calls to `pthread_join` with calls to `legup_pthreadpoll`, which, as its Pthread

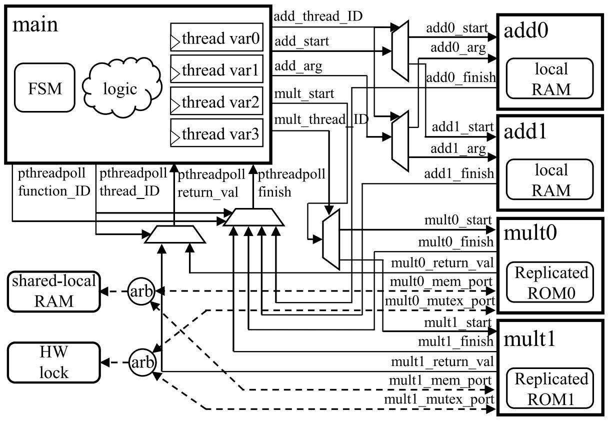

Fig. 2. Hardware-only system architecture.

counterpart, is used to wait until the hardware instance corresponding to the thread variable is finished, and retrieves the return value if needed.

The second task of the `ParallelAPI` pass is to generate hardware logic to manage threads. To perform the required bookkeeping, we propose using memories. For each different Pthread functions, we create a global variable in the LLVM IR (i.e., a single variable is created for `vector_add` in the example code, but when two different Pthread functions are called, two variables are created), which essentially acts as the thread ID counter for that function. Before each Pthread function call, the value of the thread ID (which is initialized to zero at the beginning of the program) is stored into its thread variable (i.e., threads[i] in the example code), and then, the thread ID is incremented. This allows the thread variable to contain information on which Pthread hardware instance is being used for the thread. In essence, we make use of the existing thread variable for the same purpose it is intended for in software—to keep track of threads. When there is more than one Pthread function, we assign a function ID to each of the different Pthread functions at compile time. As previously mentioned, the thread variable is used in `legup_pthreadpoll` to determine which parallel instance to be poll. Before a call to `legup_pthreadpoll`, we introduce a load instruction into the IR to retrieve the value of the thread variable. In hardware, this value is used as the *select* signal for multiplexers, which are generated to choose between the Pthread hardware instances.

Fig. 2 shows the hardware architecture created for a hardware-only system with two Pthread functions, each of which has two threads. As mentioned previously, each thread becomes an independent hardware core by default. The solid arrows in Fig. 2 show signals for calling and joining Pthread modules; the dotted arrows indicate signals for memory. To start and send arguments to a Pthread module, demultiplexers are generated to steer data to the correct parallel instance, by using the thread ID as the select signal. When the FSM reaches the state corresponding to `legup_pthreadpoll`, the caller module loads the value of the thread variable, which contains the threadID/functionID values, and outputs the values on their respective ports. Multiplexers are created to select the correct parallel instance using these values. Once the finish is asserted, the caller FSM proceeds to retrieve the return value, if one exists for the module. As can be seen in Fig. 2, we have created a lightweight hardware thread manager (registers to hold thread variables, MUX/DEMUX to select thread instances), and the actual operation of managing threads is achieved through a set of load and store operations.

The parallel hardware instances can also access memories, which can be local to a thread, or shared between multiple threads. We use a points-to analysis to determine which memories are accessed by which hardware modules. If a memory is accessed by a single module (local RAM in Fig. 2), we create it inside the module and connect it directly within, and if it is accessed by multiple modules, we create it outside the modules (shared-local RAM), with arbitration automatically created to handle contention [38]. For the arbitration, if the accessing modules execute in parallel, a round-robin arbiter is created, or if they execute sequentially (do not overlap execution), a simple OR gate is created. For each shared-local RAM, a dedicated memory port is created from the accessing module, so that independent memories can be accessed in parallel. For read-only memories, we can also replicate them across threads to localize the memories to each thread instance (replicated ROM). This helps to reduce memory contention and improve performance. When a mutex is used, we instantiate a hardware lock module, which is acquired and released through memory loads/stores [26]. Note that entire Pthreads-to-FPGA flow for a hardware-only system, including the generation of the thread managing logic, parallel hardware instances, memories, and interconnect, is completely automatic and can be done with a single command, without requiring any user changes to the input software.

A. Sharing a Hardware Core Across Threads

So far, we have described an automated synthesis flow where the number of generated parallel hardware modules, or hardware cores, is exactly equal to the number of threads in software. However, for software executing on a regular processor, a user can freely fork more threads than the number of available cores, with the OS handling the scheduling and context switching of threads, allowing multiple threads to share a core. Such time multiplexing of cores may also be important for hardware, particularly in area-constrained designs. Recognizing this, we also provide an option where a user can constrain the number of hardware cores created for a Pthread function. This can be specified with the following `Tcl` command: `set_accelerator_function "function_name" -numAccels max_number_of_instances`. With this, the hardware backend in LegUp only creates as many hardware cores for `function_name` as given by `max_number_of_instances`, and the sharing logic is created in the `ParallelAPI` pass. If a hardware instance being called is already running, the caller module waits until the instance completes its execution, then invokes the instance with the new thread.

V. PTHREADS TO PROCESSOR/PARALLEL-ACCELERATOR HYBRID SYSTEM

Although hardware can often provide speed and energy efficiency benefits compared with software, software remains

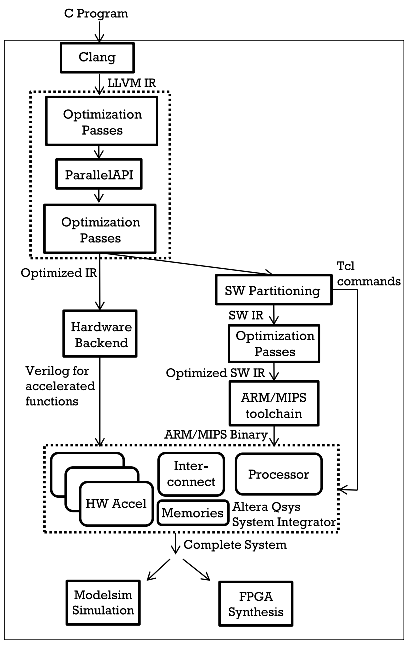

Fig. 3. Processor–accelerator hybrid flow in LegUp HLS.

more flexible than hardware, and hence, it can beneficial at times to leave some parts of the code executing in software. For instance, algorithms, which are sequential or control-intensive in nature, may not be suited to implement in hardware. To this end, we provide the processor–accelerator hybrid flow, where we automatically perform software/hardware partitioning to accelerate a portion of the code in hardware, with the remaining segments executing in software on a processor. The complete SoC, including memories and interconnect, can be automatically generated with a single command.

Fig. 3 shows the generation flow for a processor–accelerator hybrid system. The first part of the flow [shown in Fig. 3 (top left)], where the C program goes through a series of optimization passes and is given to the hardware backend, remains the same as the hardware-only flow shown in Fig. 1. This allows us to perform the identical set of compiler optimizations to the entire program with the steps specific to the hybrid flow added on as an additional procedure, allowing for easier maintainability. In the hybrid flow, the LegUp backend only compiles to hardware those functions designated for hardware acceleration. These are user-designed functions, such as Pthread functions, as well as their descendant functions. In the hardware backend, we also generate a wrapper module for each hardware accelerator, which contains Avalon interfaces (Altera's on-chip interface) to allow an accelerator to communicate with the processor and shared memories over the Avalon Interconnect [39].

In the hybrid flow, the software program must also be modified to execute the accelerated functions in hardware instead of in software. To do this, we run the `SW Partitioning` pass on the Optimized IR. This pass partitions the software portion to be executed on the processor by removing the hardware-designated functions and replacing them with wrapper functions, providing the means for the processor to communicate with the hardware accelerators over the interconnection fabric. It also assigns a unique memory-mapped address to each hardware accelerator. In the `ParallelAPI` pass, we had replaced `pthread_create` with a direct call to the Pthread function. In the `SW Partitioning` pass, we replace the Pthread function with a wrapper function, which performs memory-mapped writes over the Avalon Interconnect to transfer function arguments, and to give the start signal to the accelerator [26]. As in the hardware-only flow, we use the thread variable to keep track of threads, but in the hybrid case, we store the memory-mapped address of an accelerator to the thread variable, which is unique to each accelerator. This is again used in `legup_pthreadpoll`, which loads the memory-mapped address from the thread variable, and polls on this address until the accelerator is done, and then retrieves the return value of the accelerator if necessary.

As the `SW Partitioning` pass has knowledge of which functions are designated for hardware, it also generates `Tcl` commands, which are subsequently used by Qsys [24], Altera's system integration tool, to generate a complete system. The generated SW IR from the `SW Partitioning` pass goes through more optimizations and the final optimized SW IR is compiled with the ARM or the MIPS compiler toolchain (depending on the selected processor) to generate the software binary. Once both the software and hardware partitions have been processed, Qsys is automatically invoked, using the previously generated `Tcl` commands as input, to generate the complete SoC. Qsys instantiates the processor, memories, and hardware accelerators, and creates the Avalon Interconnect to tie all the components together. Finally, the generated system can then either be simulated with ModelSim, with the testbench and its test vectors (compiled from the software binary) automatically generated to initialize the off-chip memory, or it can be synthesized with Altera Quartus II, to produce the FPGA programming bitstream.

As in the hardware-only flow, we can also limit the number of hardware accelerators that are created for a Pthread function, to share an accelerator across multiple threads. Using the same `Tcl` command as in the hardware-only flow, `set_accelerator_function "function_name" -numAccels max_number_of_instances`, a user can set the maximum number of hardware accelerators instantiated for a Pthread function.

A. Soft MIPS

The processor–accelerator hybrid system with the soft MIPS processor provides the benefits of having a processor without requiring an SoC board. Since the MIPS is an open-source processor [40] and is implemented in FPGA soft logic, it can also be freely customized as needed. The MIPS hybrid architecture is shown in Fig. 4, which comprises the MIPS processor, hardware accelerators, on-chip caches, and off-chip DDR3 memory. The overall system architecture remains similar to what was described in [26], but we have significantly improved the accelerator memory architecture in this paper. LegUp automatically partitions memories to identify memories that are local to an accelerator, which are instantiated within the accelerator, and memories that are

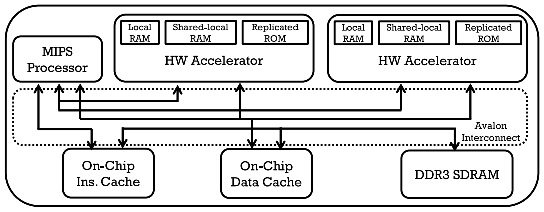

Fig. 4. Processor–accelerator architecture with MIPS.

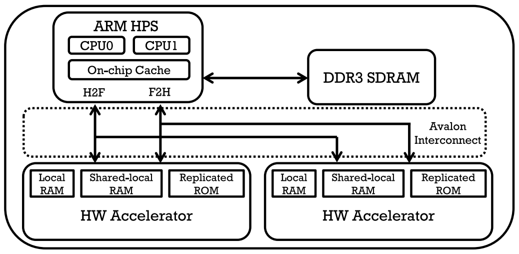

Fig. 5. Processor–accelerator architecture with ARM.

shared with the processor and/or other accelerators, which are stored in the shared memory space (on-chip data cache and off-chip memory). In the prior work, we used a simple memory partitioning algorithm, where only arrays that are local to the accelerated function (i.e., auto variables which would be stored on the stack in software) were implemented as local memories within the accelerator. Any other memories, such as global variables/arrays or local arrays of caller functions, were stored in the shared memory space. Local memories within the accelerator were accessed via a central memory controller, which limited memory accesses to *two* per clock cycle (owing to block RAMs being dual-ported).

In this paper, we have implemented a points-to analysis to determine at compile time which variables/arrays are accessed by which functions.[1] This can significantly improve performance, as all independent accelerator-side memories can be accessed in parallel, and we can also localize more memories to be within accelerators, rather than in on-chip cache/off-chip DDR3, which has a much longer memory latency on a cache miss. For example, even if an array is not declared as local to the accelerated function in the original C code, but yet the array is only accessed only by the accelerated function, it can be instantiated within the accelerator. As in a hardware-only system, a memory that is accessed by a single function (local RAM) is implemented within its accessing hardware module; a memory that is accessed by multiple functions (shared-local RAM) within an accelerator is implemented outside the accessing modules, with the arbitration created as necessary. Any memories, which are accessed by both the processor and an accelerator, or accessed by multiple accelerators, are stored in the shared memory space, but constant memories can also be replicated and localized to each hardware accelerator (replicated ROMs). All local and independent shared-local RAMs, as well as replicated ROMs, can be accessed in parallel within an accelerator.

B. Hard ARM

Fig. 5 shows the ARM processor–accelerator architecture. The ARM HPS on the Arria V SoC Development Board consists of a dual-core ARM Cortex-A9 MPCore, running at 1.05 GHz [41]. In addition to the ARM CPUs, the HPS includes 32-kB L1 on-chip instruction and data caches, a 512-kB L2 on-chip cache, and an SDRAM controller, which connects to an off-die (on-board) 1-GB DDR3 memory. A number of different interfaces are provided to communicate between the HPS and the FPGA, including the HPS-to-FPGA interface (denoted as H2F in Fig. 5), and the FPGA-to-HPS interface (denoted as F2H in Fig. 5). The H2F is an Advanced eXtensible Interface (AXI) [42] interface that allows the ARM processor to communicate with the hardware accelerators in a hybrid system. To access the shared memory space, hardware accelerators access the F2H interface (also AXI), which is connected to the accelerator coherency port (ACP), which connects to on-chip caches in the HPS to provide cache-coherent data. The HPS also provides the FPGA-to-HPS SDRAM interface (discussed in Section V-C), allowing an FPGA component to directly access the DDR3 memory without going through the cache.

The operation of the ARM hybrid system remains similar to MIPS, where the processor invokes hardware accelerators via memory-mapped operations over the Avalon Interconnect, and the accelerators access the shared memory space over the interconnect. However, since the on-chip cache resides within the HPS and has to be accessed through layers of bridges/switches, in addition to crossing clock domains (HPS runs much faster than the FPGA fabric), the memory latency is significantly higher than accessing the on-chip cache in the MIPS system, where the cache is implemented on the FPGA fabric. As for the hardware accelerators, their architecture remains the same, where they can have localized memories that can be accessed without going over the interconnect.

1) Operating System Support: Traditionally, getting an operating system to run on an FPGA system has been an arduous task. *Soft* processors are generally slow, requiring a significant engineering effort for an OS to run on them. Even when an OS successfully runs on a soft processor, its speed is typically quite slow and potentially impractical for deployment in a real system. With the introduction of hard ARM processors on FPGAs, the situation has changed. multicore ARM CPUs that run at over 1 GHz can reliably boot an OS, providing sufficient performance for embedded tasks. A user can develop and compile code on the processor itself, and even execute multithreaded code, with the OS managing threads as necessary. Having an ARM SoC FPGA with an OS opens many doors for FPGAs, where software engineers can develop, compile, and execute code in an environment they are already familiar with while having the option of accelerating a critical portion of their code on the FPGA with an HLS tool. The OS also comes preinstalled with all standard compilers and libraries.

We have set up the Linux OS flow for the Arria V SoC, where the hard ARM processor can boot the OS from an SD card. Using the OS, the ARM HPS can be used to execute

[1]The points-to analysis support was first implemented in [38], but only for hardware-only systems. In this paper, we have implemented the points-to analysis to also work for hybrid systems.

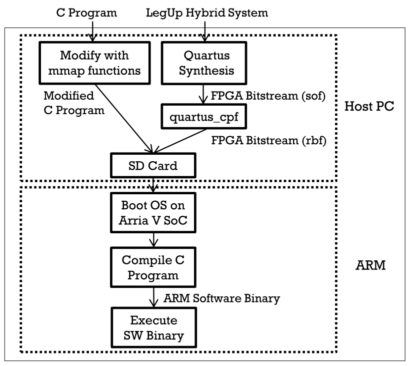

Fig. 6. ARM hybrid flow with OS.

software which works in tandem with hardware accelerators on the FPGA fabric. Note, however, that since the OS uses virtual addresses, we need to map the physical addresses of the hardware accelerators (memory-mapped addresses in the Qsys system), to virtual addresses used in the OS. To do this, we use the `mmap` Linux system call [43], which creates a mapping in the virtual address space. `mmap` can also be used to map a contiguous noncacheable region of physical memory, which is needed for performing DMA transfers between accelerators and off-chip memory [44]. An example code for using the `mmap` function to map a hardware accelerator is also provided on our wiki [45].

Fig. 6 shows the steps required to use an ARM hybrid system with an OS. First, the ARM processor–accelerator system is generated automatically with LegUp (as shown in Fig. 3), and compiled with Altera's synthesis tool, Quartus II, to generate the FPGA programming bitstream. This bitstream, in SRAM Object File format, needs to be converted to an Raw Binary File format with `quartus_cpf`, Altera's conversion tool. Then, the user needs to modify the original C code to map hardware accelerators with `mmap` functions. Both the modified C code, and the `rbf` file are copied to the SD card with the OS. All these steps are performed on the host PC. Now, the OS can be booted on the Arria V SoC from the SD card, and a serial connection program, can be used to establish the connection between the ARM and host PC. Through this connection, the user can compile the C code with `gcc` on the ARM processor, and then execute the binary, which invokes the hardware accelerators.

2) Bare Metal Support: Although having an operating system provides many benefits, as the OS handles various complicated tasks behind the scenes for a system, it can also add overheads. For a real-time application, where low jitter and deterministic operation are key factors, it can be beneficial to run the processor in bare metal (no OS). Hence, we also provide bare metal support for the ARM HPS on the Arria V SoC FPGA.

The biggest challenge to using a bare metal system is to bring up the system in the first place. Everything from writing ARM assembly code to set up the memory management unit, L1/L2 caches, and FPGA-to-SDRAM bridge, to writing a linker script that brings all these together to work with the software program, must be handled by the user. Essentially, in bare metal, much of what would be handled automatically

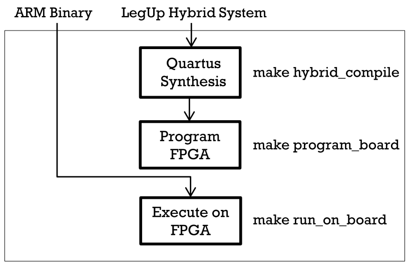

Fig. 7. ARM hybrid flow with bare metal.

by an OS must be set up manually by the user. This can require considerable engineering effort, and is a nontrivial task for both software and hardware engineers. Recognizing this, we provide with LegUp HLS a bare metal system, where the ARM HPS has been fully set up, as well as a complete toolchain that can be easily used to run applications on the ARM.

Fig. 7 shows the flow for using a bare metal ARM hybrid system. Contrary to the ARM OS flow, which currently requires the user to make minor changes to the C code to map hardware accelerators in virtual memory, the bare metal flow is completely automated. The complete SoC can be compiled, downloaded, and executed on an FPGA with three `make` targets. The generated SoC can be compiled to a programming bitstream with `make hybrid_compile`, and downloaded onto the FPGA with `make program_board`. Finally, the software binary, which has been automatically modified to call the hardware accelerators, can be downloaded onto the DDR3 memory and executed on the ARM processor (which invokes the hardware accelerators) with `make run_on_board`. The ARM cross-compiler, which creates the ARM binary, and `quartus_hps`, which programs the HPS, are all set up to be used for the Arria V SoC and executed *under the hood*. With this automated bare metal flow, one can go from C code to an ARM hybrid system running on an SoC FPGA with only a handful of commands.

C. Direct Memory Access Support

We now describe our DMA support for both the ARM and MIPS hybrid systems. First, a software library is needed to control DMA operations from the processor. We have created an intuitive, easy-to-use software library that can be used to start/reset a DMA operation, as well as cause the processor to wait until a DMA transfer is done. Our DMA library can be used by including `legup/dma.h` in the source code. For the DMA core itself, we currently use Altera's DMA Controller Core [47]. This DMA core offers basic but sufficient functionality to transfer large chunks of data in bursts.

Figs. 8 and 9 show the hybrid MIPS and ARM architectures with DMA support. For clarity, we only show the connections related to DMA operations, although the connections between the processor and hardware accelerators also exist, as were shown previously. Within a hardware accelerator, there are on-chip RAMs, which are used as input and output buffers. The accelerators consume data from their input buffers to process, and store outputs to their output buffers. A DMA core is used to fill an input buffer with data from off-chip memory,

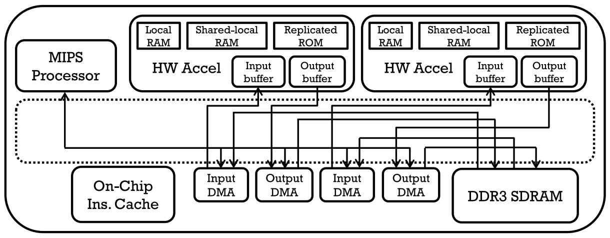

Fig. 8. Hybrid MIPS architecture with DMA support.

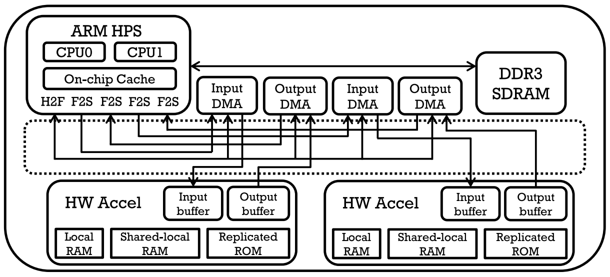

Fig. 9. Hybrid ARM architecture with DMA support.

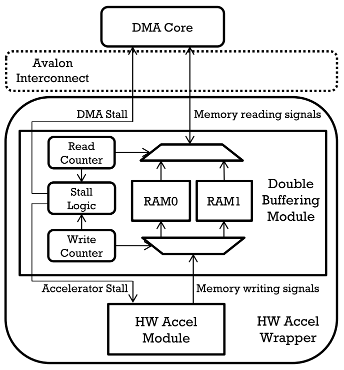

Fig. 10. Hardware accelerator with output double buffering.

and another DMA core is used to transfer data from an output buffer back to off-chip memory. Such an architecture allows concurrent input and output transfers via double buffering, which is beneficial when overlapping data transfers with accelerator computations. The F2S interfaces shown in Fig. 9 denote the FPGA-to-HPS SDRAM interfaces, which connect directly to the DDR3 SDRAM without going through the L3 interconnect (the interconnect within the ARM HPS), the ACP port, or the caches, thereby providing significantly higher memory bandwidth compared with accessing the cache. Since this direct SDRAM interface does not go through the cache, it is not cache coherent. Therefore, in the ARM hybrid systems, which use DMA, we turn OFF the data cache in the ARM HPS via the ARM startup assembly code. Similarly, the MIPS hybrid architecture shown in Fig. 8 shows that we have removed the on-chip data cache from the system.

Fig. 10 shows the architecture of a hardware accelerator where its output data are double buffered. We provide the double buffering module, which can be instantiated to use its functionality. Within the double buffering module, there are two sets of dual-ported RAMs, where the ports on one end connect to a DMA core, with the other end connecting to an accelerator module. There are also two sets of counters, one counter keeping the track of the number of writes, and the other counter keeping the track of the number of reads. When the number of writes reaches the depth of the RAM, it switches the RAM being written to, and when the number of reads reaches the depth, the DMA reads are also steered to the other RAM. The double buffering module also provides stall logic, which is important if the data rate for reads is not as fast as writes, and vice versa. Currently, the instantiations of the DMA cores and the double buffering module are not automatically handled by LegUp and must be done by the user, but we have plans to automate this in the future.

VI. Experimental Study

In this section, we study the performance, area, and energy-efficiency of the three types of systems described in this paper: 1) the hardware-only system without a processor; 2) the MIPS processor–accelerator hybrid system; and 3) the ARM processor–accelerator hybrid system. For each type of system, we investigate a number of different hardware architectures.

For a hardware-only system, we examine three different architectures. First, a single-threaded software program (without Pthreads) is compiled to sequential hardware (denoted as Arch. 1 in Section VI-B). Then, we parallelize the same program with Pthreads using three threads (denoted as Arch. 2), which gets compiled to three concurrently operating hardware cores. Finally, we pipeline each one of the three cores (denoted as Arch. 3p) with loop pipelining. Note that due to the structure of the benchmarks, not all benchmarks were pipelinable, and hence for the results shown in Section VI-B, an architecture with the p postfix includes results for only those benchmarks which could be pipelined (four out of six benchmarks).

For the MIPS/ARM hybrid systems, Arch. 3p (multi-threaded and pipelined) uses DMA transfers to move data in and out of hardware accelerators (double buffering is not yet used in this case). For these systems, we also investigate an additional architecture, called the processor-only system. Denoted as Arch. 0 and Arch. 0p (a subset of results including only the pipelinable benchmarks), the entire program is executed in software on the MIPS/ARM processor (without hardware accelerators). Note that all these systems (both processor-only and hybrid) run bare metal (without an OS).

We also examine the performance and energy consumption of one of the benchmarks, Black–Scholes, when running on an ARM hybrid system, versus when running purely in software on the ARM processor, as well as on two different x86 processors, the Intel Xeon E5-1650 and the Intel i7-4770K, where all these systems (ARM, ARM hybrid, and x86) are running Linux OS on their processors. The Intel Xeon E5-1650 is a 32-nm six-core (12 cores with Hyper-Threading [48]) CPU, running at 3.2 GHz (3.8 GHz with TurboBoost) with 2-MB L2 and 12-MB L3 caches. It is running Ubuntu 14.04 and has 32 GB of DDR3 memory running at 1333 MHz. The Intel i7-4770K is a 22-nm four-core (eight cores with Hyper-Threading) CPU, running at 3.5 GHz (3.9 GHz with Turbo-Boost) with 1-MB L2 and 8-MB L3 caches. It is also running Ubuntu 14.04 and has 32 GB of DDR3 memory running at 1600 MHz. We investigate the performance and energy consumption of the Black–Scholes benchmark when running on one, two, and three threads, on the ARM processor-only, ARM hybrid, and x86 systems. For the ARM hybrid systems, we pipeline the threads and use DMA transfers with double buffering. For the x86 systems, we also investigate using as many threads as the number of logical cores (number of cores seen with HyperThreading).

A. Benchmarks and Measurement Methodologies

We use a total of six benchmarks, each of which has a number of different variants. For hardware-only systems, we create two versions of each benchmark, a sequential version and a parallelized version with Pthreads. For hybrid systems, we create an additional variant, which uses the DMA functions from our DMA software library. For running the Black–Scholes benchmark on the ARM hybrid systems with Linux, we use the mmap functions described previously. The six benchmarks are described in the following.

1) *Black–Scholes:* Estimates the price of the European-style options. It uses Monte Carlo simulation to compute the price trajectory for an option using random numbers. Computations are done in fixed point.
2) *Dfdiv:* Adopted from the CHStone benchmark suite [49], it computes double-precision floating-point division using 64-bit integers.
3) *Dfsin:* Adopted from the CHStone benchmark suite [49], it computes the sine function for double-precision floating-point numbers using 64-bit integers.
4) *Gaussian Filter:* Used to filter out noise by convolving the Gaussian filter with an image.
5) *Hash:* Runs four different hashing algorithms.
6) *Mandelbrot:* An iterative mathematical benchmark which generates a fractal image.

Each benchmark includes built-in inputs and golden outputs, with the computed result checked against the golden output at the end of the program to verify correctness. For hardware-only systems, input/output data resides in on-chip memory, and for hybrid systems, they are stored in off-chip DDR3 memory. Out of the six benchmarks, Dfdiv and Dfsin were not pipelinable. We synthesize each hardware system using Quartus 15.0 targeting the Altera Arria V SoC FPGA (5ASTFD5K3F40I3) to obtain area and critical path delay (*Fmax*) numbers. For hardware-only and MIPS systems, we use ModelSim to extract the total number of execution cycles and compute the total execution time (wall-clock time) as the product of the execution cycles and the postrouted clock period. To obtain power data, we use the postsynthesis timing simulation to generate a Value Change Dump file with ModelSim, which is used by the Quartus PowerPlay Power Analyzer to produce power numbers. To get the execution time on ARM systems running bare metal (processor-only and hybrid), we use the performance monitoring unit on the ARM processor [50] to get the processor cycle count, and divide this by the processor's clock speed, 1.05 GHz, to calculate the wall-clock time. We start the counter before starting the hardware functions, and stop after they have returned, which also includes DMA configuration and transfer times for those systems with DMA, but does not include the time to verify individual elements of transferred output data on the processor. For measuring power on the ARM systems, we use the power monitor on the Arria V SoC Development Kit Board [51], which measures real-time power consumption of both the HPS and the FPGA fabric via their power rails.

We also compare the Black–Scholes benchmark when running on a single thread, as well as on multiple threads, on the ARM processor-only, ARM hybrid, and x86 systems, with all processors running Linux OS. We have made our best efforts to make this comparison as fair as possible. Before measurements, we reboot the systems to start from a clean slate and kill any user-spawned processes. To compile the benchmark running purely in software, we use gcc with -O3 optimization and the -pthread flag (on both ARM and x86). To obtain execution time, we use the gettimeofday function, a Linux function that can be used to get the current wall-clock time. We call it before pthread_create and after pthread_join (which includes DMA transfer times for the hybrid case) to get the total execution time. With the OS, if the total runtime is very short, the thread startup and stop times can be a significant portion of the runtime, which puts the processor-only systems at a disadvantage. To avoid this, we increase the total number of Black–Scholes simulations done to the maximum that could fit in the 1-GB DDR3 memory, which was around 600 MB of output data (each simulation output is stored in memory to be verified on the processor). Also with the OS, runtime is not always deterministic, so we execute the benchmark 100 times on all platforms and take the geometric mean. On the x86 processors, we set the governor to performance, which allows the CPUs to run at maximum speed with TurboBoost. To measure the effect of TurboBoost on CPU frequency, we use turbostat, a Linux command-line utility which reports real-time stats, such as frequency and temperature [52]. In addition, to get more consistent results over runs, we also use taskset [53], a Linux command which allows binding a process to specific cores to avoid the process from jumping between different cores. We use the first three cores on both the Xeon and the i7 CPUs. To measure power on the x86 processors, we use Intel's Performance Counter Monitor [54], which also has a utility to measure power consumption.

B. Results

Table I shows the geometric mean results over all the benchmarks for the sequential (Arch. 1/1p), multithreaded (Arch. 2/2p), and multithreaded and pipelined (Arch. 3p) architectures of hardware-only systems. We show four different types of metrics: 1) performance, which includes total execution time (in ms), total clock cycles, and Fmax (in MHz); 2) area, which includes the number of adaptive logic modules (ALMs), registers, DSPs, and M10Ks (Altera's memory blocks, which can hold 10 kbits); 3) power, which includes static and dynamic power (in mW), as well as the sum of the two; and 4) finally, efficiency, which includes total energy consumption (in mJ), calculated as the product of total execution time and total power, and area-delay product. To calculate the area-delay product, we first calculate the total chip area by using the data from [55] and [56], which gives

TABLE I

GEOMETRIC MEAN RESULTS FOR HARDWARE-ONLY SYSTEMS

	Performance			Area				Power			Efficiency	
Architecture	**Time (ms)**	**Cycles**	**Fmax**	**ALMs**	**Registers**	**DSPs**	**M10Ks**	**Static Power**	**Dyn. Power**	**Total Power (mW)**	**Energy (mJ)**	**Area-delay Product**
Arch. 1	2.06	342,846.46	166.10	3,931.72	7,987.11	48.86	5.83	1,328.75	48.26	1,393.94	2.88	25,358.72
Arch. 1p	2.70	481,973.16	180.53	2,880.20	5,909.52	43.12	7.03	1,328.77	45.64	1,396.86	3.73	25,839.86
Arch. 2	0.79	121,838.05	153.76	10,609.17	22,698.89	122.05	10.32	1,329.24	111.13	1,452.28	1.15	25,454.88
Arch. 2p	1.00	168,556.69	169.33	7,060.69	16,032.12	98.28	6.82	1,329.03	93.37	1,432.48	1.43	22,398.69
Arch. 3p	0.02	3,840.87	175.78	9,406.79	24,251.33	98.28	17.18	1,329.65	159.86	1,501.21	0.03	620.43
Arch. 2 / Arch. 1 Ratio	0.384 (2.60×)	0.355 (2.81×)	0.926	2.698	2.842	2.498	1.771	1.000	2.303	1.042	0.400 (2.50×)	1.00 (1.00×)
Arch. 3p / Arch. 1p Ratio	0.008 (122.19×)	0.008 (125.49×)	0.974	3.266	4.104	2.280	2.443	1.001	3.503	1.075	0.009 (113.69×)	0.024 (41.65×)
Arch. 3p / Arch. 2p Ratio	0.022 (45.56×)	0.023 (43.89×)	1.038	1.332	1.513	1.000	2.521	1.000	1.712	1.048	0.023 (43.47×)	0.028 (36.10×)

TABLE II

GEOMETRIC MEAN RESULTS FOR MIPS PROCESSOR–ACCELERATOR HYBRID SYSTEMS

	Performance			Area				Power			Efficiency	
Architecture	**Time (ms)**	**Cycles**	**Fmax**	**ALMs**	**Registers**	**DSPs**	**M10Ks**	**Static Power**	**Dyn. Power**	**Total Power (mW)**	**Energy (mJ)**	**Area-delay Product**
Arch. 0	67.22	8,618,499.64	128.21	9,629.00	12,444.00	6.00	76.00	1,339.35	325.94	2,218.01	149.10	1,811.04
Arch. 0p	69.70	8,935,681.09	128.21	9,629.00	12,444.00	6.00	76.00	1,339.35	325.94	2,218.01	154.59	1,877.74
Arch. 1	3.36	420,025.54	124.93	14,683.81	23,363.61	55.20	80.13	1,340.18	466.13	2,314.93	7.78	139.86
Arch. 1p	4.22	527,468.28	125.12	13,711.87	21,310.13	49.39	79.22	1,339.96	443.61	2,290.03	9.65	164.47
Arch. 2	1.27	159,685.16	125.83	21,610.78	38,805.89	129.30	86.34	1,342.50	711.96	2,579.92	3.27	78.90
Arch. 2p	1.60	196,517.80	122.86	17,357.50	30,753.96	105.44	85.52	1,341.40	604.20	2,457.33	3.93	81.86
Arch. 3p	0.26	28,444.26	110.98	20,621.39	39,706.97	105.44	255.40	1,345.27	987.15	2,839.69	0.73	18.03
Arch. 1 / Arch. 0 Ratio	0.050 (19.99×)	0.049 (20.52×)	0.974	1.525	1.878	9.199	1.054	1.001	1.430	1.044	0.052 (19.16×)	0.077 (12.95×)
Arch. 2 / Arch. 0 Ratio	0.019 (52.97×)	0.019 (53.97×)	0.981	2.244	3.118	21.550	1.136	1.002	2.184	1.163	0.022 (45.54×)	0.044 (22.96×)
Arch. 3p / Arch. 0p Ratio	0.004 (271.93×)	0.003 (314.15×)	0.866	2.142	3.191	17.574	3.361	1.004	3.029	1.280	0.005 (212.40×)	0.010 (104.15×)
Arch. 2 / Arch. 1 Ratio	0.377 (2.65×)	0.380 (2.63×)	1.007	1.472	1.661	2.343	1.077	1.002	1.527	1.114	0.421 (2.38×)	0.564 (1.77×)
Arch. 3p / Arch. 1p Ratio	0.061 (16.44×)	0.054 (18.54×)	0.887	1.504	1.863	2.135	3.224	1.004	2.225	1.240	0.075 (13.26×)	0.110 (9.12×)

the tile area for each type of blocks, including ALM, DSP, and memory blocks. With the total area accounting for the different types of blocks, we multiply this with the total wall-clock time to obtain the area-delay product for each architecture. The last three lines of Table II show the relative ratios between the architectures.

Looking at the ratios between the architectures, we see significant improvements in performance for Arch. 2 and especially for Arch. 3p, over Arch. 1. With three parallel cores, Arch. 2 shows 2.81× and 2.6× speedups in clock cycles and wall-clock time, respectively, with area increasing by 2.70 times for ALMs, 2.84 times for registers, 2.50 times for DSPs, and 1.77 times for M10Ks. Dynamic power consumption increases by 2.3 times, although total power increases by only 4.2%. In terms of energy efficiency, we see an improvement of 2.5 times, but the area-delay product basically stays the same. Memory bandwidth is a crucial factor for performance, especially for pipelined hardware, which can require new input data every clock cycle. Thus, for Arch. 3p, we partition the input/output memories and replicate constant memories across threads, to minimize stalls caused by memory contention. With memory partitioning/replication and three pipelined cores operating concurrently, Arch. 3p shows vast speedups over Arch. 1p, where clock cycles is improved by 125.5 times, and wall-clock time is improved by 122.2 times. The dynamic power consumption increases by 3.5 times (total power by 7.5%), but the large improvements in performance leads to 113.7 times better energy efficiency and 2.4% area-delay product (41.7 times improvement). Comparing Arch. 3p with Arch. 2p, we see a 71.2% increase in dynamic power, but also 43.5 times better energy efficiency with 2.8% area-delay product (36.1 times improvement). Thus, with proper memory organization and sufficient memory bandwidth, we show that pipelining with Pthreads can give vast performance/energy benefits.

Table II shows the geometric mean results for the MIPS processor-only system (Arch. 0/0p), as well as for the hybrid systems (Arch. 1/1p, 2/2p, 3p). The relative ratios are shown in comparison with Arch. 0/0p, in addition to Arch. 1/1p. When comparing with Arch. 0/0p, all architectures exhibit significant improvements in performance, energy-efficiency, and area-delay product. With three concurrent cores (Arch. 2), we see a speedup of 53 times in wall-clock time, while being 45.5 times more energy efficient. Fmax stays relatively constant, as the critical path is mostly limited by the on-chip cache and the interconnect in the processor system. As expected, the most notable improvements come from Arch. 3p, which uses pipelining with DMA transfers (without double buffering). Compared with Arch. 0p, Arch. 3p shows 271.9 times speedup and 212.4 times energy efficiency, with 104.2 times better area-delay product. With three parallel pipelined accelerators, DSP usage increases notably by 17.6 times, since the processor-only system only uses six DSP blocks. Other area elements also increase by from 2.14 times to 3.36 times, and with this, Fmax drops by 13.4% and total power consumption increases by 28%. In comparison with Arch. 1p, Arch. 3p shows improvements of 16.4 times, 13.3 times, and 9.1 times in wall-clock time, energy-efficiency, and area-delay product, respectively. Area also increases by from 1.5 times to 3.2 times for the different types of blocks, with total power consumption going up by 24%. Again, we show that Pthreads with pipelining, together with high memory bandwidth provided by DMA transfers, offers significant benefits over sequential hardware, as well as over software executing on a soft MIPS processor.

Table III shows the geomean results for ARM processor-only and hybrid systems running bare metal. The power consumption, obtained via the power monitor, is shown in terms of HPS power (which includes power for the HPS core, HPS I/O, HPS DDR3, and HPS internal/peripheral devices [57]), FPGA power (which includes power for FPGA core/clock, FPGA I/O, and FPGA internal/peripheral devices), and total power (the sum of the two). We do not show area-delay product results, since the HPS area is unknown. All clock cycle numbers shown on this table are in terms of processor cycles,

TABLE III
GEOMETRIC MEAN RESULTS FOR ARM PROCESSOR–ACCELERATOR HYBRID SYSTEMS

	Performance			Area				Power			Efficiency
Architecture	**Time (ms)**	**Cycles**	**Fmax**	**ALMs**	**Registers**	**DSPs**	**M10Ks**	**HPS Power**	**FPGA Power**	**Total Power (mW)**	**Energy (mJ)**
Arch. 0	1.50	1,578,439.36	1,050.00	1,650.00	2,551.00	0.00	4.00	1,754.73	340.73	2,095.59	3.15
Arch. 0p	2.20	2,305,313.96	1,050.00	1,650.00	2,551.00	0.00	4.00	1,756.03	339.87	2,096.10	4.60
Arch. 1	3.29	3,453,671.54	140.96	6,187.96	12,420.47	48.86	7.25	1,739.42	447.47	2,187.77	7.20
Arch. 1p	4.24	4,450,387.83	146.78	5,173.58	10,337.21	43.12	6.90	1,754.34	452.35	2,207.30	9.36
Arch. 2	1.67	1,754,443.41	125.66	13,904.07	30,795.08	122.05	13.14	1,755.17	556.22	2,314.79	3.87
Arch. 2p	1.78	1,869,232.14	127.29	10,417.79	23,748.90	98.28	12.54	1,749.68	511.65	2,262.07	4.03
Arch. 3p	0.07	73,167.36	144.57	15,501.36	33,486.58	98.28	121.43	1,767.26	716.93	2,484.79	0.17
Arch. 1 / Arch. 0 Ratio	2.188 (0.46×)	2.188 (0.46×)	0.140	3.750	4.869	48.858	1.812	0.991	1.313	1.044	2.284 (0.44×)
Arch. 2 / Arch. 0 Ratio	1.112 (0.90×)	1.112 (0.90×)	0.125	8.427	12.072	43.116	3.284	1.000	1.632	1.105	1.228 (0.81×)
Arch. 3p / Arch. 0p Ratio	0.032 (31.51×)	0.032 (31.51×)	0.144	9.395	13.127	122.050	30.357	1.006	2.109	1.185	0.038 (26.58×)
Arch. 2 / Arch. 1 Ratio	0.508 (1.97×)	0.508 (1.97×)	0.891	2.247	2.479	2.498	1.812	1.009	1.243	1.058	0.537 (1.86×)
Arch. 3p / Arch. 1p Ratio	0.016 (60.82×)	0.016 (60.82×)	0.985	2.996	3.239	2.280	17.596	1.007	1.585	1.126	0.019 (54.03×)

and the execution times are obtained by dividing the number of clock cycles by the processor speed, 1.05 GHz. The Fmax results shown for Arch. $1 \sim 3p$ are the frequencies of the PLLs that are driving hardware accelerators' clocks. The HPS processor-only system (Arch. 0) consumes a small amount of FPGA area due to its peripherals, such as JTAG UART, which is used by the processor to communicate with host PC (using `printf`), as well as the interconnect. Note that the FPGA fabric also consumes about 340 mW in this case, even when all processing is done on the HPS. With the hard ARM processor running at over 1 GHz, the processor-only system can achieve fairly good performance. With this, in addition to the limited memory bandwidth of accessing the high-latency on-HPS cache through the ACP port from hardware accelerators, Arch. 1 shows a slow down of 2.19 times compared with Arch. 0. With 4.4% more total power consumption (31.3% more FPGA fabric power), energy consumption is also 2.28 times of Arch. 0. This trend continues to Arch. 2, which exhibits 11.2% more runtime than Arch. 0, with 10.5% more total power (63.2% more FPGA fabric power) and 22.8% more energy consumption. This again shows that memory bandwidth is critical to performance (even more so than the hardware itself in some cases) and that nonpipelined hardware with limited memory bandwidth cannot outperform the hard ARM processor. On the other hand, Arch. 3p, which has three pipelined hardware accelerators with direct access to DDR3 via DMA, shows significant improvements over the ARM processor system. The architecture shows 31.5 times speedup in wall-clock time with 26.6 times better energy efficiency. The hybrid systems with LegUp-generated hardware accelerators can significantly outperform a 1-GHz ARM processor in both performance and energy efficiency. When compared with Arch. 1p, Arch. 3p is 60.8 times faster and 54 times more energy efficient.

One may notice that the area numbers in Table III are considerably higher than Table I. We attribute much of this increase to the Qsys-generated Avalon Interconnect, in addition to some area consumed by the ARM HPS. Although Qsys generates a flexible interconnect, it has a high area overhead, whereas the LegUp-generated interconnect for hardware-only systems is much simpler, and therefore smaller. Additional area is consumed in Arch. 3p for ARM due to the DMA cores, which also contribute to the significant increase in M20K usage (DMA cores use on-chip RAMs for buffering data).

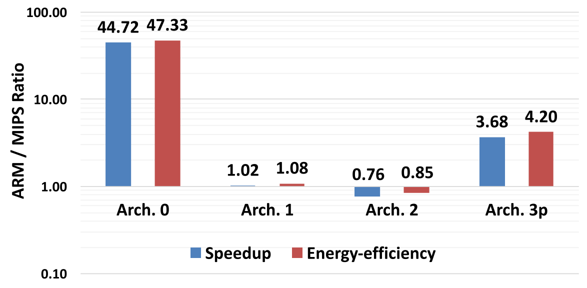

Fig. 11. Speedup/energy-efficiency ratios for ARM over MIPS.

With all architectures implemented on the same FPGA, using the same set of benchmarks, it can be interesting to contrast the results of the ARM versus the MIPS systems. Fig. 11 shows the relative speedup and energy-efficiency ratios (a log scale) of ARM versus MIPS for each type of architecture (MIPS is used as the baseline in each case). Observe that for Arch. 0, where all computations are performed in software, the ARM processor shows significant speedup and energy-efficiency benefits compared with the MIPS (44.72 times and 47.33 times, respectively), showcasing the advantage of the hardened processor. When most computations are moved to a single accelerator in Arch. 1, the systems exhibit similar results. When computations are parallelized with Pthread accelerators in Arch. 2, the ARM hybrid system shows a slow down, as the memory latency, hence the number of cycles the accelerators are stalled due to memory contention arising from concurrent accelerators accessing the cache, is larger for the ARM hybrid architecture. However, the memory bandwidth is improved in Arch. 3p, since the HPS offers multiple FPGA-to-HPS SDRAM interfaces, allowing the parallel accelerators to access off-chip memory concurrently. With this, the ARM hybrid systems for Arch. 3p show 3.68 times speedup and 4.20 times energy-efficiency improvement over the MIPS hybrid systems.

With hardened processors on FPGAs, we believe that an increasing number of FPGA applications will run an OS. Therefore, we also analyzed the performance overhead of running an OS versus bare metal, by executing all Arch. 0 benchmarks from Table III on the ARM with an OS. We observed that the OS had virtually no performance overhead (~2% higher wall-clock time, on average) compared with bare metal. These results are consistent with [58], which showed that for a real-time application running on a Cyclone V

TABLE IV
RESULTS FOR BLACK–SCHOLES ON ARM PROCESSOR-ONLY, ARM HYBRID, AND x86 ARCHITECTURES

Architecture	Time (s)	Power (W)	Energy (J)
ARM SW 1T	50.717	2.128	107.921
ARM SW 2T	25.418	2.327	59.159
ARM SW 3T	25.870	2.337	60.453
ARM Hybrid 1T	1.184	2.301	2.725
ARM Hybrid 2T	0.642	2.529	1.622
ARM Hybrid 3T	0.484	2.683	1.297
Xeon SW 1T @ 3.75 GHz	3.312	55.214	182.878
Xeon SW 2T @ 3.75 GHz	1.668	69.699	116.273
Xeon SW 3T @ 3.65 GHz	1.144	79.234	90.640
i7 SW 1T @ 3.9 GHz	2.735	24.096	65.909
i7 SW 2T @ 3.9 GHz	1.372	26.504	36.355
i7 SW 3T @ 3.8 GHz	0.944	35.459	33.476

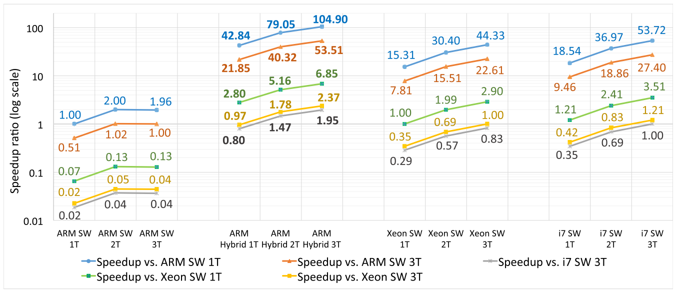

Fig. 12. Speedup results for Black–Scholes on ARM processor-only, ARM hybrid, and x86 architectures.

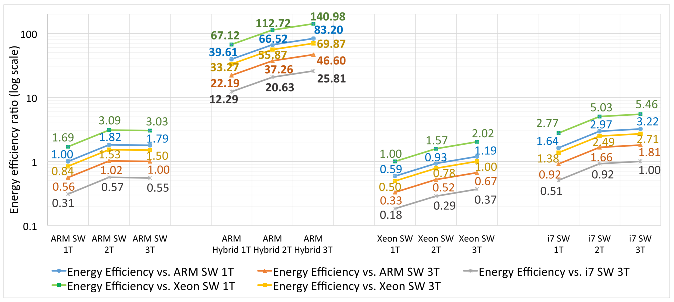

Fig. 13. Energy-efficiency results for Black–Scholes on ARM processor-only, ARM hybrid, and x86 architectures.

ARM SoC FPGA, a bare metal solution performed no better than an OS-based solution.

Table IV shows the performance, power, and energy consumption results of the Black–Scholes benchmark when executing purely in software on the ARM (denoted as ARM SW), the Intel Xeon E5-1650 (Xeon SW), and the Intel i7-4770K (i7 SW), as well as when running on the ARM hybrid architecture (ARM Hybrid). On each platform, we increase the number of threads from one (1T), to two (2T), and to three (3T). All architectures are running Linux on their processors and all hybrid systems are using DMA transfers with double buffering. Table IV also shows the frequencies of the x86 processors with TurboBoost, measured with `turbostat` while the benchmark is being executed. All frequency values fall within the range of their turbo bins provided by Intel, which indicate the frequency increases that can be achieved with TurboBoost for each number of active cores [59]. In general, Table IV shows that for each architecture, as the number of threads is increased, the performance improves, with increasing power consumption. The results remain similar, however, for ARM SW 2T and ARM SW 3T, as the ARM processor has a dual-core CPU.

To make the comparisons between the architectures easier, we plot their relative ratios on Figs. 12 and 13. The x-axis shows the different architectures, with increasing number of threads from left to right, and the y-axis shows their ratios in logarithmic scale. Each plotted line shows the ratio values of a single architecture when compared across all other architectures on the x-axis, and for readability, we only plot five different architectures (out of 12) as shown in the legends. Fig. 12 shows the speedup results for the Black–Scholes benchmark, where the hybrid architecture speedups are highlighted in bold. First, when comparing ARM Hybrid 3T with i7 SW 3T (running at 3.8 GHz), Xeon SW 3T (running at 3.65 GHz), and ARM SW 3T (running at 1.05 GHz), we see the speedups of 1.95 times, 2.37 times, and 53.51 times, respectively. It is worth noting that ARM Hybrid 3T shows 104.9 times speedup compared with ARM SW 1T. This speedup is much larger than what was shown in Table III for Arch. 3p/0p mainly due to double buffering. Even with two concurrent hardware accelerators (ARM Hybrid 2T), we can still outperform the triple threaded case of software, by 1.47 times (i7 SW 3T), 1.78 times (Xeon SW 3T), and 40.32 times (ARM SW 3T). The LegUp-generated hardware systems can surpass the performance of Intel CPUs which are running much faster in clock speeds (120 MHz versus ~4 GHz).

Energy efficiency is also another strength of FPGAs, and this demonstrated in Fig. 13, where all hybrid systems exhibit at least an order of magnitude improvement in energy efficiency compared with all SW systems. Compared with i7 SW 3T, Xeon SW 3T, and ARM SW 3T, ARM Hybrid 3T shows 25.81 times, 69.87 times, and 46.6 times better energy efficiency, respectively. The Xeon processor consumes a considerable amount of power, so the ARM Hybrid 3T is 140.98 times more energy efficient than Xeon SW 1T. Even with a single accelerator (ARM Hybrid 1T), we observe improvements of 12.29 times and 33.27 times compared with i7 SW 3T and Xeon SW 3T, and 24.19 times (calculated from Table IV) and 67.12 times when compared with i7 SW 1T and Xeon SW 1T. Again, this demonstrates that we can achieve significant improvements in energy efficiency with LegUp-generated ARM hybrid systems.

One may question, however, that there are unutilized cores on the Intel CPUs which can simply be used by forking more threads. To investigate this, we modified the Black–Scholes benchmark to use as many threads as the number of logical cores (number of cores with HyperThreading) for each Intel CPU. Table V shows the result for the Xeon CPU when using 12 threads, and for the i7 CPU with eight threads, as well as their relative ratios compared with ARM Hybrid 3T. By utilizing more cores, both CPU frequencies with TurboBoost have dropped as expected, in accordance with their turbo bin values. In terms of total execution time, ARM Hybrid 3T still outperforms the Intel CPUs, by 4%

TABLE V
RESULTS FOR BLACK–SCHOLES ON x86 PROCESSORS WHEN USING AS MANY THREADS AS THE NUMBER OF CORES

Architecture	Time (s)	Power (W)	Energy (J)
Xeon SW 12T @ 3.5 GHz	0.504	110.560	55.740
vs. ARM Hybrid 3T	0.484 (1.04×)	2.68 (41.21×)	1.30 (42.97×)
i7 SW 8T @ 3.65 GHz	0.578	45.990	26.580
vs. ARM Hybrid 3T	0.484 (1.20×)	2.68 (17.14×)	1.30 (20.49×)

compared with Xeon SW 12T, and by 20% compared with i7 SW 8T. The power consumption of both CPUs also increase with more utilized cores, and thus, we still see significant energy-efficiency improvements of 42.97 times compared with Xeon SW 12T, and 20.49 times compared with i7 SW 8T. Hence, the LegUp ARM hybrid system still provides substantial benefits even when all Intel CPU cores are utilized. Moreover, additional hardware accelerators can be instantiated in the ARM hybrid case as well, since the ARM Hybrid 3T only consumes about 10% of logic and 7% of DSP blocks on the Arria V SoC FPGA. We expect that having more accelerators will increase performance until it is limited by memory bandwidth (DMA to/from off-chip DDR3).

VII. CONCLUSION AND FUTURE WORK

In this paper, we presented a framework that can compile Pthreads to parallel hardware cores, constituting a hardware-only system, or a processor–accelerator hybrid system. For using a hybrid system, the user can choose to use a soft MIPS processor, or a hard ARM processor, and for targeting the ARM processor, the user can choose to run bare metal (no OS), or a Linux OS.

The experimental results showed that the multithreaded and pipelined architectures (with three hardware cores) can achieve significant improvements in speed and energy efficiency when compared with the sequential hardware-only system, and when compared with MIPS/ARM processor-only systems. LegUp-generated ARM hybrid systems also outperformed multithreaded software running on the Intel i7 and Xeon CPUs, in addition to having more an order of magnitude better energy efficiency.

In conclusion, we showed that one can write Pthread software, run it on a regular processor, then using the same software (the same number of threads, minimal code changes in the case of using OS or DMA), compile it to a fully working FPGA system, and achieve higher performance and much better energy efficiency, even when compared with real-world processors. HLS has been around for decades, and one of the main barriers to its adoption has been its ease-of-use. One can easily generate a hardware core, but it is still mostly up to the user to handle the integration of the core into a fully working system. With this paper, we simplify that effort and expose hardware spatial parallelism to software, so that a nonhardware engineer can create a complete multicore system from software. This can be flexibly realized with a purely hardware system, with a soft MIPS processor-parallel accelerator system, or with a hard ARM processor-parallel accelerator system (with and without an OS), which is not possible with any other HLS tool.

In the future work, we would like to automate the instantiations of the DMA cores and the double buffering module to create a more streamlined flow for using our DMA features.

REFERENCES

[1] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, “High-level synthesis for FPGAs: From prototyping to deployment,” *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 30, no. 4, pp. 473–491, Apr. 2011.

[2] W. Zuo *et al.*, “Improving high level synthesis optimization opportunity through polyhedral transformations,” in *Proc. ACM FPGA*, Feb. 2013, pp. 9–18.

[3] Y. Liang *et al.*, “High-level synthesis: Productivity, performance, and software constraints,” *J. Electr. Comput. Eng.*, vol. 2012, p. 14, Jan. 2012. [Online]. Available: https://www.hindawi.com/journals/jece/2012/649057/cta/

[4] R. Nane *et al.*, “A survey and evaluation of FPGA high-level synthesis tools,” *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 35, no. 10, pp. 1591–1604, Oct. 2015.

[5] M. Abdelfattah *et al.*, “Gzip on a chip: High performance lossless data compression on FPGAs using OpenCL,” in *Proc. Int. Workshop OpenCL*, 2014, pp. 4:1–4:9.

[6] D. Singh *et al.*, “High-level design tools for floating point FPGAs,” in *Proc. Int. Symp. FPGAs*, Feb. 2015, pp. 9–12.

[7] A. Canis *et al.*, “LegUp: High-level synthesis for FPGA-based processor/accelerator systems,” in *Proc. ACM/SIGDA FPGA*, Feb. 2011, pp. 33–36.

[8] C. Pilato and F. Ferrandi, “Bambu: A modular framework for the high level synthesis of memory-intensive applications,” in *Proc. FPL*, Sep. 2013, pp. 1–4.

[9] R. Nane *et al.*, “DWARV 2.0: A CoSy-based C-to-VHDL hardware compiler,” in *Proc. IEEE FPL*, Oslo, Norway, 2012, pp. 619–622.

[10] *Vivado High-Level Synthesis*. Accessed on Aug. 30, 2016. [Online]. Available: http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[11] *Altera SDK for OpenCL*. Accessed on Aug. 30, 2016. [Online]. Available: https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html

[12] *Catapult High-Level Synthesis*. Accessed on Aug. 30, 2016. [Online]. Available: https://www.mentor.com/hls-lp/catapult-high-level-synthesis/

[13] *Impulse Accelerated Technologies*. Accessed on Aug. 30, 2016. [Online]. Available: http://www.impulseaccelerated.com

[14] Bluespec Inc. [Online]. Available: http://www.bluespec.com

[15] *Cyberworkbench*. Accessed on Aug. 30, 2016. [Online]. Available: http://www.nec.com/en/global/prod/cwb/index.html

[16] D. Hansen *et al. Calypto White Paper: Designing ASIC IP at Higher Level Abstraction*. Accessed on Aug. 30, 2016. [Online]. Available: https://www.mentor.com/hls-lp/success/qualcomm-inc

[17] *Intel Xeon Phi Product Family: Product Brief*. Accessed on Aug. 30, 2016. [Online]. Available: http://www.intel.com/content/www/us/en/high-performance-computing/high-performance-xeon-phi-coprocessor-brief.html

[18] *Nvidia Tesla GPU Accelerators*. Accessed on Aug. 30, 2016. [Online]. Available: http://international.download.nvidia.com/pdf/kepler/TeslaK80-datasheetpdf

[19] B. Barney. *POSIX Threads Programming*. [Online]. Available: https://computing.llnl.gov/tutorials/pthreads/

[20] *The OpenMP API Specification for Parallel Programming*. Accessed on Aug. 30, 2016. [Online]. Available: http://openmp.org/wp/

[21] *The Open Standard for Parallel Programming Heterogeneous System*. Accessed on Aug. 30, 2016. [Online]. Available: https://www.khronos.org/opencl/

[22] *CUDA Parallel Computing Platform*. Accessed on Aug. 30, 2016. [Online]. Available: http://www.nvidia.ca/object/cuda_home_new.html

[23] *Xilinx: Xcell Journal, Issue 86*. [Online]. Available: http://www.xilinx.com/publications/archives/xcell/Xcell86.pdf

[24] *Qsys—Alteras System Integration Tool*. Accessed on Aug. 30, 2016. [Online]. Available: https://www.altera.com/products/design-software/fpga-design/quartus-prime/features/qts-qsys.html

[25] *Vivado IP Integrator: Accelerated Time to IP Creation and Integration*. Accessed on Aug. 30, 2016. [Online]. Available: http://www.xilinx.com/publications/prod_mktg/vivado/Vivado_IP_Integrator_Backgrounder.pdf

[26] J. Choi, S. Brown, and J. Anderson, "From software threads to parallel hardware in high-level synthesis for FPGAs," in *Proc. IEEE FPT*, Dec. 2013, pp. 270–277.

[27] *Altera SoC*. Accessed on Aug. 30, 2016. [Online]. Available: https://www.altera.com/products/soc/overview.html

[28] *Expanding the All Programmable SoC Portfolio*. Accessed on Aug. 30, 2016. [Online]. Available: http://www.xilinx.com/products/silicon-devices/soc.html

[29] *MicroBlaze Soft Processor Core*. Accessed on Aug. 30, 2016. [Online]. Available: http://www.xilinx.com/products/design-tools/microblaze.html

[30] *Nios II Processor: The World's Most Versatile Embedded Processor*. Accessed on Aug. 30, 2016. [Online]. Available: https://www.altera.com/products/processors/overview.html

[31] G. Stitt and F. Vahid, "Thread warping: A framework for dynamic synthesis of thread accelerators," in *Proc. CODES+ISSS*, Sep. 2007, pp. 93–98.

[32] E. Anderson *et al.*, "Enabling a uniform programming model across the software/hardware boundary," in *Proc. IEEE FCCM*, Apr. 2006, pp. 89–98.

[33] A. Ismail and L. Shannon, "FUSE: Front-end user framework for O/S abstraction of hardware accelerators," in *Proc. IEEE FCCM*, May 2011, pp. 170–177.

[34] E. Lubbers and M. Platzner, "A portable abstraction layer for hardware threads," in *Proc. IEEE FPL*, Sep. 2008, pp. 17–22.

[35] *SDSoC Development Environment*. Accessed on Aug. 30, 2016. [Online]. Available: http://www.xilinx.com/products/design-tools/software-zone/sdsoc.html

[36] *SDSoC Environment User Guide*. Accessed on Aug. 30, 2016. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2/ug1027-intro-to-sdsoc.pdf

[37] *SDAccel Development Environment*. Accessed on Aug. 30, 2016. [Online]. Available: http://www.xilinx.com/products/design-tools/software-zone/sdaccel.html

[38] J. Choi, S. D. Brown, and J. H. Anderson, "Resource and memory management techniques for the high-level synthesis of software threads into parallel FPGA hardware," in *Proc. IEEE FPT*, Dec. 2015, pp. 152–159.

[39] *Avalon Interface Specifications*, Altera, San Jose, CA, USA, 2015.

[40] *The Tiger 'MIPS' Processor*, Univ. Cambridge, Cambridge, U.K., 2010. [Online]. Available: http://www.cl.cam.ac.uk/teaching/0910/ECAD+Arch/mips.html

[41] *Altera Arria V SoC*. Accessed on Aug. 30, 2016. [Online]. Available: https://www.altera.com/products/soc/portfolio/arria-v-soc/overview.html

[42] *AMBA Specifications*. Accessed on Aug. 30, 2016. [Online]. Available: https://www.arm.com/products/system-ip/amba-specifications.php

[43] *Linux Programmer's Manual: MMAP*. Accessed on Aug. 30, 2016. [Online]. Available: http://man7.org/linux/man-pages/man2/mmap.2.html

[44] *Altera SDK for OpenCL: Cyclone V SoC Development Kit Reference Platform Porting Guide*. Accessed on Aug. 30, 2016. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/ug_aocl_c5soc_devkit_platform.pdf

[45] *LegUp wiki*. Accessed on Aug. 30, 2016. [Online]. Available: http://legup.org/wiki

[46] *Picocom(8)—Linux Man Page*. Accessed on Aug. 30, 2016. [Online]. Available: http://linux.die.net/man/8/picocom

[47] *Embedded Peripherals IP User Guide*. Accessed on Aug. 30, 2016. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/ug_embedded_ip.pdf

[48] *Intel Hyper-Threading Technology*. Accessed on Aug. 30, 2016. [Online]. Available: http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html

[49] 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," *J. Inf. Process.*, vol. 17, pp. 242–254, Oct. 2009.

[50] *Chapter 11: Performance Monitoring Unit, Cortex-A9 Technical Reference Manual (Revision: r3p0)*. ARM, Cambridge, U.K., 2010.

[51] *Arria V SoC Dev Kit User Guide*. Accessed on Aug. 30, 2016. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/ug_av_soc_dev_kit.pdf

[52] *Ubuntu Manuals: Turbostat*. Accessed on Aug. 30, 2016. [Online]. Available: http://manpages.ubuntu.com/manpages/xenial/man8/turbostat.8.html

[53] *Taskset*. Accessed on Aug. 30, 2016. [Online]. Available: http://linuxcommand.org/man_pages/taskset1.html

[54] *Intel Performance Counter Monitor—A Better Way to Measure CPU Utilization*. Accessed on Aug. 30, 2016. [Online]. Available: www.intel.com/software/pcm

[55] H. Wong *et al.*, "Comparing FPGA vs. Custom CMOS and the impact on processor microarchitecture," in *Proc. 19th ACM/SIGDA Int. Symp. FPGAs*, 2011, pp. 5–14.

[56] R. Rashid *et al.*, "Comparing performance, productivity and scalability of the TILT overlay processor to OpenCL HLS," in *Proc. IEEE FPT*, Dec. 2014, pp. 20–27.

[57] *Arria V SoC Develop.Board Reference Manual*, accessed on Aug. 30, 2016. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/rm_av_soc_dev_board.pdf

[58] C. Phoon *et al. Bare-Metal, RTOS, or Linux? Optimize Real-Time Perform. with Altera SoCs*, accessed on Aug. 30, 2016. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01245-optimize-real-time-performance-with-altera-socs.pdf

[59] *Intel Turbo Boost Technology Frequency Table*, accessed on Aug. 30, 2016. [Online]. Available: http://www.intel.com/content/www/us/en/support/processors/000005523.htm

Jongsok Choi (S'06) received the B.A.Sc. degree from the University of Waterloo, Waterloo, ON, Canada, in 2009, and the M.A.Sc. degree in computer engineering and the Ph.D. degree in computer engineering, with a focus on the automatic synthesis of multithreaded software to parallel hardware in high-level synthesis, from the University of Toronto, Toronto, ON, Canada, in 2012 and 2016, respectively.

Dr. Choi received the Natural Sciences and Engineering Research Council of Canada Alexander Graham Bell Canada Graduate Scholarship for his Ph.D. studies.

Stephen D. Brown (M'90) was the Director of Research and Development with the Altera Toronto Technology Center, Toronto, ON, Canada, from 2000 to 2008. He is currently a Professor with the University of Toronto, Toronto. He is also the Director of the Worldwide University Program with Altera, Toronto, ON, Canada. He is an Author of over 70 scientific publications and the Co-Author of three textbooks, the *Fundamentals of Digital Logic with VHDL Design* (New York: McGraw-Hill, 2004), the *Fundamentals of Digital Logic with Verilog Design* (New York: McGrawHill, 2002), and *Field-Programmable Gate Arrays* (Kluwer Academic, 1992). His current research interests include CAD algorithms, field-programmable gate arrays, VLSI technology, and computer architecture.

Jason H. Anderson (S'96–M'05) received the B.Sc. degree in computer engineering from the University of Manitoba, Winnipeg, MB, Canada, and the M.A.Sc. and Ph.D. degrees in electrical and computer engineering from the University of Toronto (U of T), Toronto, ON, Canada.

He joined the Field-Programmable Gate Array Implementation Tools Group, Xilinx, Inc., San Jose, CA, USA, in 1997, where he was involved in placement, routing, and synthesis. He is currently an Associate Professor with the Department of Electrical and Computer Engineering, U of T, and holds the Jeffrey Skoll Endowed Chair in software engineering. He has authored over 70 papers in refereed conference proceedings and journals, and holds 27 issued U.S. patents.