Gallatin, et al. Twill: A hybrid microcontroller-FPGA framework for parallelizing single-threaded C programs. *IEEE International Symposium on Parallel and Distributed Processing 2014*.

Twill: A Hybrid Microcontroller-FPGA Framework for Parallelizing Single-Threaded C Programs

Doug Gallatin, Aaron Keen, Chris Lupo, and John Oliver
Computer Engineering Program
California Polytechnic State University
San Luis Obispo, California 93407
Email: {dgallati, akeen, clupo, jyoliver}@calpoly.edu

***Abstract*—Increasingly System-On-A-Chip platforms which incorporate both microprocessors and re-programmable logic are being utilized across several fields ranging from the automotive industry to network infrastructure. Unfortunately, the development tools accompanying these products leave much to be desired, requiring knowledge of both traditional embedded systems languages like C and hardware description languages like Verilog. We propose to bridge this gap with Twill, a truly automatic hybrid compiler that can take advantage of the parallelism inherent in these platforms. Twill can extract long-running threads from single threaded C code and distribute these threads across the hardware and software domains to more fully utilize the asymmetric characteristics between processors and the embedded reconfigurable logic fabric. We show that Twill provides a significant performance increase on the CHStone benchmarks with an average 1.63 times increase over the pure hardware approach and an increase of 22.2 times on average over the pure software approach while in general decreasing the area required by the reconfigurable logic compared to the pure hardware approach.**

I. Introduction

Increasingly it is becoming common for Field Programmable Gate Array (FPGA) manufacturers to embed microprocessors within the FPGA fabric. This allows developers on such systems to pick and choose which parts of their application require the speedups achievable by being implemented in hardware while maintaining a faster development/debug cycle for majority of the (nontime-critical) code.

The development cycle for these kinds of hybrid systems has thus been writing assembly, C or C++ code for the microcontroller and HDL code for the surrounding FPGA logic framework and then manually specifying the interface between the two code sections. While this paradigm gives the developer flexibility and control, the complexity of the HW/SW interface leads to many hard-to-debug errors in all but the simplest of systems. In turn this leads to longer development cycles and requires more experienced, specialized developers which often pushes many potential products to use less efficient solutions.

A. Related Works

Several tools have been developed recently to assist with this problem. Historically there have been two different approaches. One is to provide a run-time system of synchronization and communication primitives to the developer so as to support inter processor-FPGA fabric communications in an abstract manner. For example, ReconOS [1] and hThreads [2] implement a real-time operating system (RTOS) where OS primitives such as queues, semaphores, and the scheduler are accessible in a uniform manner from both HW and SW "threads". In contrast, works by [3] and [4] provide uniform APIs only for calling "functions" defined either in HW or in SW from either the FPGA logic or from the microprocessor. All of these run-time systems abstract away some or all of the communication between the HW and SW but still require the developer to write in both C and HDL and to explicitly set up any parallelism.

The other approach is to implement or modify a compiler that both partitions the input code into HW/SW modules and then generates the communication and synchronization channels to tie everything together. Examples of these include LegUp [5,6], Spark [7], and Liquid Metal [8]. Liquid Metal introduces a new Java-based object-oriented language that allows the programmer to interact with object instances across the HW/SW divide but requires the programmer to keep track of which objects are where.

LegUp and Spark both implement a compiler/translator for traditional C programs. LegUp was originally only an HDL translator but has recently added limited support for calling functions across the HW/SW divide. They have a basic automatic heuristic but encourage the programmer to annotate each function with whether that function should be implemented in HW or in SW. LegUp does not do any sort of Thread-Level Parallelism (TLP) but does implement a modulo-scheduler for Instruction-Level Parallelism (ILP). Also, LegUp does not provide any primitives other than the function call for synchronization and communication which makes it extremely difficult for the programmer to implement truly parallel code.

In contrast, Spark started with a similar system to LegUp and then focused on implementing code optimizations in order to achieve speedup. With complete control over the hardware, they were able to implement several different speculative-based optimizations with very little overhead. However, they focused almost entirely on ILP parallelization techniques at the expense of TLP parallelization.

B. Twill

Twill is designed as a compiler to take single-threaded C code as input, extract long running threads from that C code, transform some of the threads into hardware, and then provide a runtime communication system for a hybrid CPU/FPGA System-On-A-Chip.

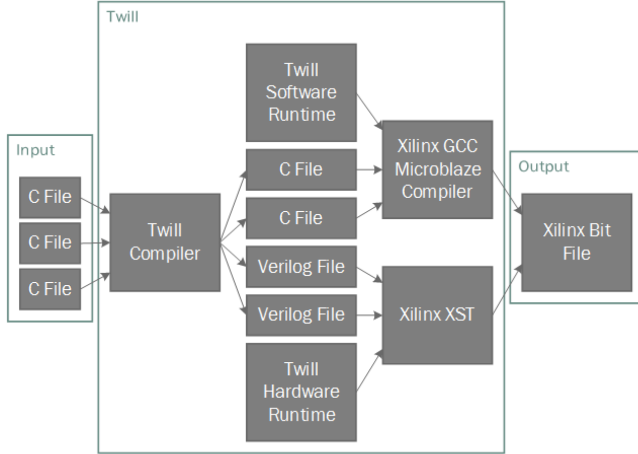


Fig. 1: Twill Overview

In this way Twill is able to take advantage of both ILP and TLP that may be present in the original source. Twill builds upon a great deal of previous work: Twill uses a modified version of Distributed Software Pipelining (DSWP) as first presented in [9] in order to find and extract long running threads. Twill relies upon LegUp for finding ILP in the extracted threads and for translating those threads into HDL. Finally, it uses a custom runtime system/RTOS heavily influenced by the hThreads project [2].

The major contribution of Twill is to integrate algorithms for ILP and TLP parallelism into an environment suitable for small, low-powered embedded systems. It combines the strengths of the hThreads/ReconOS systems with the abstraction provided by LegUp and Spark while exploiting a higher degree of the parallelism inherent in the input program. Thus, it is able to give very large performance speedups for these kinds of hybrid embedded systems without requiring the programmer to have any knowledge of HDL.

The remainder of our paper is organized as follows:

An overview of Twill is presented in Section II. Section III details the runtime architecture of Twill while Twill's compiler implementation is described in IV. In Section V we discuss Twill's performance results and finally in Section VI we conclude.

II. Overview of Twill

Twill conceptually consists of three different parts: the compiler, the software runtime system, and the hardware runtime system. An overview of how these fit together can be seen in Figure 1.

1) Twill Compiler: The Twill compiler is described in detail in Section IV. Internally it is implemented as multiple LLVM [10] transform passes including a custom version of the DSWP algorithm [9] and then uses LegUp [5] to translate the hardware portions into Verilog. It also sets parameters for the statically defined primitives in both the software and hardware runtime systems.

While conceptually the Twill compiler could be ANSI C compliant, it currently has the same limitations on the input C files as LegUp: no recursive functions or function pointers. While this simplifies our implementation, Section VI-A expands on how Twill could be extended to support these two constructs.

2) Twill Software Runtime: The Twill software runtime is written in C and assembly. It contains an API for interfacing the processors with the hardware runtime. The Twill compiler generates C code for the processors with calls to these APIs in order to perform initialization, thread management, synchronization, and communication. It is described in more detail in Section III.

3) Twill Hardware Runtime: The Twill hardware runtime written in Verilog is based heavily off of the hThreads project [2]. It provides synchronization and communication primitives for the software and hardware threads. The generated Verilog modules from the Twill compiler include "calls" to the various hardware primitives to provide synchronization and communication with the other Verilog modules and the software threads running on the processors. Section III describes in depth the implementation details of the Twill hardware runtime.

III. Run-Time Architecture

Twill's runtime system is heavily influenced by the hThreads project [2]. The runtime system has several primitives: semaphores, queues, software threads, hardware threads, and a simple scheduler. All of the primitives are statically configured at compile time with the exception of software threads which can be dynamically created. Semaphores, queues, the scheduler, and the hardware threads are all implemented in the FPGA logic in Verilog. Hardware threads are able to interact with semaphores, queues, and the processor's memory without interrupting the processor while software threads have minimal API wrappers to interact with the hardware primitives. The entire architecture overview is shown in Figure 2.

The following subsections discuss the individual primitives after describing the bus addressing system.

A. Bus Architecture

There are two main communication busses in Twill's runtime system that tie all of the primitives together. The first bus, the Module Bus, is the main communications link between all of the primitives used for passing messages. The second bus,

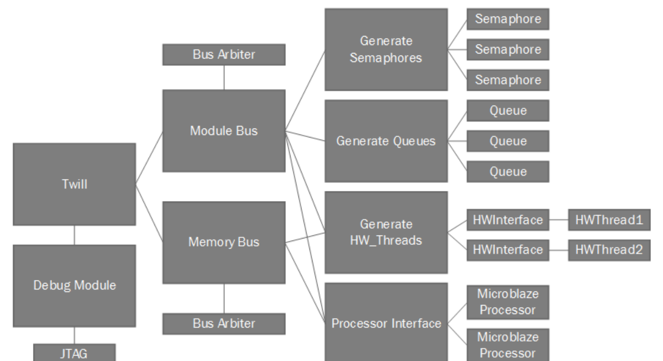


Fig. 2: Twill Run-Time Hardware Architecture Overview

the Memory Bus, is tied to each of the hardware threads and the processor interface module and gives the hardware threads access to the processor's memory space. The two busses are hierarchical with the Generate blocks shown in Figure 2 used to decrease the combinatorial logic at each stage allowing for higher clock frequencies.

Both busses work on a message passing model. Each primitive is assigned a unique address for the busses. When a primitive needs to send a message, it signals to the bus arbiter and for each clock cycle the bus arbiter will specify which primitive has control over the bus along with that primitive's bus message. This is designed in such a way that if there is no contention for the bus among the primitives, a primitive's signal will be acknowledged and its message available on the very next clock cycle. Thus, the bus has a latency of one clock cycle and a throughput of one message per clock cycle.

The bus arbiter is implemented as a modified priority decoder which always gives priority to the processor if it is signaling and then gives priority to any primitive sending a message to the processor and finally gives priority to the primitive who has been signaling for the longest number of clock cycles. This is because the processor interface with hardware tends to be the critical path since the processor is slower at executing instructions. Furthermore, since the processor generally takes longer to perform a task than the pure hardware threads, it tends to signal the bus less frequently and when it does the system is designed such that the processor's pipeline should not be stalled at all waiting for the hardware primitive to respond.

A message on the main message bus consists of the destination address, the 3-bit message operation, and a 32-bit data field. The destination address is variably sized depending on the number of primitives. There are five operations: `give`, `take`, `start`, `stop`, and `ack`. Most primitives only accept a subset of the operations and the effect of the operations vary depending on what type of primitive is located at the destination address. The primitive specific effects from these operations are described in the following sections.

The memory bus uses the same model and timing characteristics of the main bus but is used solely to allow the hardware threads to read and write processor memory. A write takes one cycle while a read takes two cycles assuming no bus contention. One hardware thread may read/write to this bus at once completely asynchronously from what the processor is doing. Writes to memory from either the processor or from the hardware threads take two cycles to appear in the other domain.

B. Semaphores

The semaphore primitives are basic counting semaphores. Each may have a different max count and starting counter. A `give` message to the semaphore will raise the semaphore while a `take` message corresponds to a lower. The data part of the message specifies how many times to raise or lower the semaphore. The semaphore will respond to the calling primitive's address with an `ack` message when that primitive has successfully taken the semaphore. When the semaphore is not locked the `ack` message will occur immediately on the next clock assuming no bus contention. If the semaphore's counter is already at zero then the semaphore will wait until a `give` message is received. The semaphore will then send `ack` messages first to the processor and then to the primitive that has been waiting the longest. In general, it is safe to send `take` messages to the semaphore from any primitive although it is not safe to send multiple `take` messages from the same primitive without receiving a corresponding `ack` message in between each message.

With the above architecture, the sending thread will be blocked for one cycle for a raise operation and a minimum of two cycles for a lower operation.

C. Queues

The queue primitives are first-in-first-out (FIFO) queues. Each may have a different max length and be either 1 bit, 8 bits, 16 bits, or 32 bits wide. The queues are asynchronous but assume that a single primitive is enqueueing data and a (potentially different) single primitive is dequeueing data. Thus a semaphore or other synchronization method between primitives must be used if more than one primitive is enqueueing the data at once or more than one primitive is attempting to dequeue the data at once. A `give` message to the queue enqueues the message's data field to the queue. An `ack` message will subsequently be sent back to the sender. A `take` message will cause the queue to send an `ack` message back to the sender with the dequeued value. Internally, the queues are implemented as a circular buffer with one more data element than the queue can hold. On enqueue operations, an `ack` message will be sent back to the sender immediately as long as the final data slot in the queue is empty. When the size+1 data slot is filled, an `ack` message will not be sent until a dequeue operation is performed. In this way the sending primitive is stalled if the queue is full. Similarly, if the queue is empty then the queue will only send the `ack` message for a dequeue operation after a `give` message is received.

The synchronization overhead of enqueueing or dequeueing from a queue is thus a minimum of two cycles assuming no bus contention.

D. Hardware Threads

Hardware threads are user written or auto-generated HDL code that perform the desired computations. They have a simple interface to the HWInterface modules which deal with the specifics of communicating over the busses. For the hardware thread to perform any action, it sets the specified function code and the desired target along with any data parameters and then sends a pulse on a signal wire to the HWInterface. The HWInterface module then will latch in all the data and make the appropriate call. Note that the desired function code is just the equivalent to an enum where each function call has its own entry. The function code does not correspond to bus operation but uniquely specifies whether to perform an enqueue, dequeue, raise, lower, load, store, etc. operation. Furthermore, the desired target is not the same as the address but rather an index into a virtual array of the OS primitive implied by the function code. For example, passing zero as the desired resource to a raise call will raise the first semaphore while passing zero as the desired resource to an enqueue call will enqueue to the first queue. Multiple calls to different

primitives may be made at once; the only constraint is that only one call may be initiated per cycle.

Each call will “return” to the hardware thread by the hardware interface specifying the code and resource on the return wires along with any data that might have been returned on the incoming data wires. In this way one function call per cycle may return to the hardware thread. The operations that “return” immediately on the next clock cycle assuming no bus contention are memory store, semaphore raise, start thread, and stop thread. Operations that take multiple cycles are memory load, semaphore lower, enqueue, and dequeue. The HWInterface can also signal to the hardware thread that another thread started or stopped it asynchronously to any pending requests.

The HWInterface module connecting the HWThread modules to the Generate HW Threads block in Figure 2 is responsible for managing all of the simultaneous requests and their response states. It is designed in such a way as to not add latency between the hardware thread's operation request and sending messages out on the bus and thus the hardware thread has the minimum cycles listed in the other sections of synchronization overhead.

There are several special system hardware threads that handle some system-related tasks. The first is the I/O manager which is connected to the serial port and all of the external interrupt pins, reset signals, LED's, and switches. Other threads can send messages to this thread to interact with the I/O ports. Interrupts are forwarded with one clock cycle latency to the appropriate handler either in hardware or on the processor.

The second special system hardware thread is the timing thread which is used to time all of the cycle counts referenced in Section V. The final special hardware thread is the scheduler. The scheduler is a simple round-robin scheduler for the software threads which can handle threads in both blocked and waiting states. Every period it will interrupt the processor with the new SW thread ID to switch to. It also snoops on the message bus looking for the active thread to become blocked in order to switch out threads. Since all of this logic is in hardware, the only critical-path cost on the processor is a single context-switch unlike traditional schedulers which require two context switches in addition to running the actual scheduling algorithm.

E. Processor Interface

The processor interface provides the method of connecting a variable number of Microblaze processors to the two busses. It is split into two parts: Verilog code that creates the actual connections and a C library that runs on each of the processors.

The C library provides function APIs such as Enqueue(), RaiseSemaphore(), and StartThread(). It also provides an interrupt controller that interfaces with the I/O hardware thread to pass interrupt sources to the proper SW thread's interrupt routine.

The communication between the C library and the hardware module is implemented using a single Microblaze Stream. Streams are built into the Microblaze processor and act very similarly to the hardware queues described above. There are two instructions in the Microblaze ISA, `put` and `get`, that each take two cycles for their data to be transfered into or out of the FPGA logic. When the streams are full or empty they will stall the processor if the corresponding `put` or `get` instruction is executed. It takes two `put`/`get` instructions to pass a message to or from the processor interface. Thus since the processor interface is designed to mask as much of the hardware overhead as possible it takes five cycles for the processor to complete any operation with any of the hardware primitives. Because of the way the message bus priority works, the worst latency possible with processor messages is $4 + n$ cycles where n is the number of processors attached to the system.

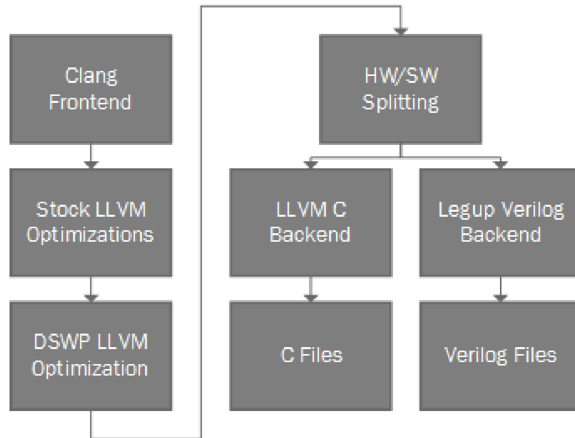


Fig. 3: Twill Compiler Tool Flow

The hardware processor interface module has only one address on the main bus no matter how many processors there are. It internally queues and interleaves the processor operations, simulating any multiple requests to the same primitive from the processors. This was done to reduce the already large overhead of having the processor communicate with the hardware primitives.

The processor interface also manages the memory between the processors and the hardware threads. Each processor has its own copy of the memory and the hardware threads share another copy. A simple write-update coherency scheme is used simply because of the small size of the memories used in the project. If the memories were larger a more sophisticated coherency scheme could be used if needed with little adverse effect on the overall architecture.

IV. Compiler Architecture

The runtime system was designed in order to optimize the DSWP algorithm [9] constraints and to simplify the Twill compiler pass. The compiler is a multi-stage patchwork of other work and custom compiler passes as seen in Figure 3.

The first step is the standard LLVM [10] tool-flow. LLVM 2.9 is used in order to have the LLVM IR directly compatible with the LegUp toolchain. After this a custom LLVM transformation pass which implements a modified version of the DSWP algorithm is run.

A. DSWP

The DSWP algorithm [9] conceptually pipelines loops by building a complete Program Dependence Graph (PDG) of the loop and then partitioning it into separate threads such that data is forwarded in only one direction between the threads. This technique was chosen as the main source of TLP parallelism because the original authors discovered that it became more efficient as the simplicity of the processing cores increased and because the required low-level and low synchronization-cost queues were relatively easy to create with control over the hardware.

There are several algorithmic differences between our implementation and the implementation described in Ottoni et al.'s original paper [9] that we discuss below.

1) Partitioning: Ottoni et al.'s implementation of DSWP used a very simple greedy heuristic where they essentially sorted each Strongly-Connected Component (SCC) in the PDG based on the sum of the estimated computation times for all the instructions in each SCC and then filled each partition from the list taking the longest running SCCs first. Note that not all SCCs are available in the list but only those whose dependencies have already been assigned a partition in order to ensure a pipeline between the threads would form.

In Twill, we assign two different weights to each SCC. The hardware weight consists of the sum of the estimated cycle·area products that would result by translating each instruction into hardware. The software weight consists of the estimated number of cycles required to execute the instruction on a Microblaze processor. A sorted list of SCCs is maintained very similarly to the original algorithm where only SCCs with no dependencies are kept on the list. Whenever a new partition is started, the total hardware weight is compared to the total software weight of each SCC currently on the list. The new partition is then designated as a hardware or software thread accordingly and filled with the smallest SCCs on the list. A targeted percentage passed by the developer for the division of work between the hardware and software domains is used to decide when to start a new partition.

2) Function Calls: Probably the biggest difference is that our implementation of DSWP operates on the function level rather than on the loop-level. While pipelining code outside of loops is of questionable benefit, it allows us to implement a key extension to the original DSWP algorithm. The original algorithm treated function calls as a single large-latency instruction and thus would not pipeline any functions outside of the function containing the manually designated loop to pipeline. By extending the pipeline to the function level, our implementation treats function calls as zero latency instructions and then sets up a special dependence so that a sub-tree of threads will pipeline the called function. This sub-tree of threads will reuse the existing threads in the current pipelining when there is no recursion involved.

Therefore in our implementation, each function contains a "master" thread and zero or more "slave" threads. The thread that the call instruction is partitioned into becomes the master thread for the new function and is responsible for passing the arguments and receiving the returned result value. The other threads call the remaining slave versions of the function. All of the slave threads for that function do not accept any arguments for the function and instead will create standard `enqueue`/`dequeue` instruction pairs with the master thread only if the partitioner gives instructions to the slave thread requiring those arguments.

Thus when a function call is found, the pipeline is rebuilt for that function based off of the thread with the call instruction and then the old pipeline resumes once the function call has finished. This does create situations where data must flow against the direction of the original pipeline which puts the queue latency on the critical path of the execution. It also potentially causes multiple versions of the same function to have to be translated into each hardware thread which increases the FPGA area required. To solve both problems, we move each function's master and slave threads into separate threads as long as the various call-sites to each function cannot execute at the same time. Within a single function, this is determined by a simple conservative heuristic which requires all call sites to have an unbroken chain of dependencies between them in order to be considered non-overlapping. Semaphores are used to ensure the function is indeed non-overlapping if the function has call-sites in multiple functions. In practice most of the time functions that do have overlapping calls tend to be simple functions that the partitioner will not partition anyways and thus the above two problems are avoided a majority of the time.

Furthermore, this method of resolving function calls potentially switches which partitions of a function are placed into software and hardware. Thus the function calls are resolved as the last step in the custom DSWP algorithm and the entire algorithm is iterated upon with different partitioning target percentage and roles for the partitions of the particular function that is called.

3) Conditional Control Dependencies: Since LLVM IR is in Single-Static-Assignment (SSA) form, some of the additional artificial conditional control dependencies introduced in the original paper are not implemented. The SSA form and its PHI nodes ensures that these scenarios cannot occur. However, there is an additional problem that LLVM's implementation of PHI nodes introduces. In LLVM, the PHI nodes may assign a constant based off of the control flow entering the block. An example of this problem is illustrated in Figure 4. The problem occurs when the partition that contains the PHI node does not have any instructions in one of the preceding basic blocks: BB2, BB3, or BB4. In this case according to [9] those basic blocks would not be present in the partition and thus the resulting threads will not be correct. Intuitively, the PHI node is control dependent on the branches in BB1 and BB3 but because of how LLVM handles PHI nodes it is not possible to forward the result of the branches using `enqueue`/`dequeue` instructions. Instead, we create a pair of fake dependencies between the PHI node and the branch instruction of every block that is associated with a constant. These dependency pairs can be seen in the dashed lines in Figure 4. This essentially forces the problematic branches and the PHI node to be on the same partition.

4) Loop Matching: Another difference in our implementation is how loops are handled. In the original implementation only one loop was handled in each program. Since functions can have an arbitrary number of loops arranged in an arbitrary fashion, care must be taken to ensure the `enqueue` and

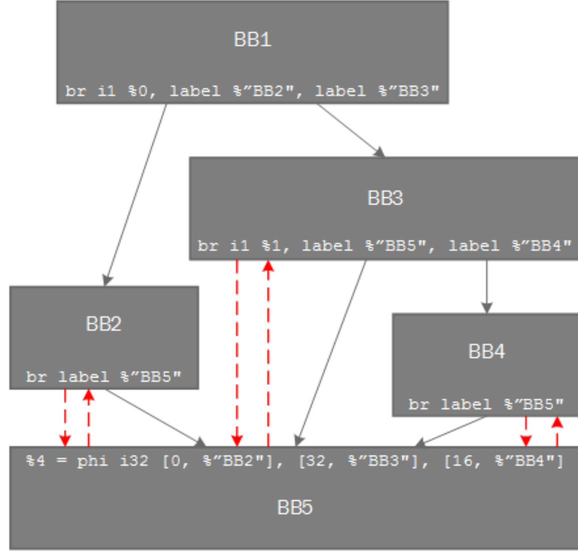
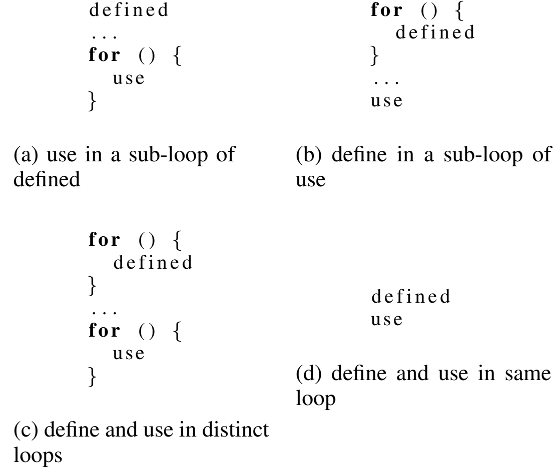


Fig. 4: PHI Node Example Control Flow Graph: Solid edges represent the control flow while dashed edges represent the fake dependencies



(a) use in a sub-loop of defined

(b) define in a sub-loop of use

(c) define and use in distinct loops

(d) define and use in same loop

Fig. 5: Enqueue/Dequeue Loop Matching Cases

dequeue instructions are matched between loops properly. For each enqueue/dequeue pair we look at the loop structure and find the lowest loop in the original function that contains both the instruction whose result needs to be enqueued in the master thread (defined) and the instruction that uses the defined instruction in the slave thread (use). At this point there are four cases shown in Figure 5. Figure 5 (d) shows the basic case where the loops are well matched. Trivially, the enqueue instruction is inserted directly after the defined instruction while the dequeue instruction is inserted directly before the use instruction.

For the case shown in Figure 5 (a), the enqueue instruction is inserted after the defined instruction while the enqueue instruction is inserted at the end of all of the use instruction's loop preheader blocks. Similarly, for the case in Figure 5 (b) the dequeue instruction is inserted directly before the use instruction while the enqueue instruction is inserted at the beginning of all of the defined instruction loop's exit blocks. In the case shown in Figure 5 (c) the enqueue instruction is inserted in all of the exit blocks while the dequeue instruction is inserted in all of the preheader blocks. Note that this will create asymmetric numbers of enqueue/dequeue instructions but will ensure that for any given control flow each loop iteration will have matching instruction numbers.

Furthermore, for every enqueue/dequeue pair a simple flow algorithm is run on the lattice formed by the common dominator and post-dominator nodes of the use and defined instructions to ensure that every enqueue is matched with a corresponding dequeue. The flow algorithm places dummy enqueue and dequeue instructions as required such that enqueue instructions are as close to the dominator node as possible while dequeue instructions are as close to the post-dominator node as possible.

Even after doing a flow adjustment this leads to some edge cases where naively doing the above will break the code. Whenever the preheader blocks have successors other than the loop header or when the exit blocks have predecessors other than blocks within the loop control flow is broken. In these cases, special basic blocks not present in the original function must be created between the block outside of the loop and the blocks inside the loop. The dequeue/enqueue instruction is then placed into this block and the branches are adjusted accordingly.

Another case where doing the above will break the code is if the use instruction is a PHI node and the dequeue instruction would be placed directly before the PHI node. In this case a new basic block not present in the original function is created on the control path between the basic block the PHI node is in and the basic block the defined instruction is in. The dequeue instruction is then placed in this basic block.

5) Homogeneous Threads: The final major difference between the original DSWP implementation and our modified implementation is that since the threads are not going to be run on homogeneous cores, the thread partitioner creates uneven partitions. It also ensures that all allocations and deallocations across all of the function calls are on a single special thread since a single thread must be in charge in order to keep the heap in sync.

B. HW/SW Splitting

After the DSWP transformation is finished, the generated threads must be split from the single LLVM IR file into HW and SW components. This stage generates a different set of stand-alone LLVM IR for each individual HW thread and SW thread based off of the results from the DSWP partitioner. Currently the special memory management thread is forced to be in software to take advantage of the standard C library's malloc/free although it would be straightforward to implement these two functions in hardware to allow hardware threads to manage the memory and to relax the requirement that all memory allocations must be on one thread. In practice, for

media applications there are very few memory allocations inside the main computation loop which makes this limitation less problematic.

The only other special requirement for the split is that the master for the main function is always implemented in the software so the processor drives the entire program execution which is required for many SOC systems. After these two threads have been assigned, the larger partition sizes are prepared for the hardware translation while the smaller partitions are put onto any remaining processor cores. Only one thread for each processor is assigned unless the threads can be demonstrated not to overlap in execution time so that context switches are avoided.

Once the individual stand-alone LLVM IR files for each thread are generated, they are passed into the LLVM C backend for the software threads or into the LegUp Verilog backend for the hardware threads.

C. LegUp Modifications

We modified LegUp in several areas to interface with the Twill hardware runtime. First, the signals needed to interface with Twill's hardware runtime system were added to all generated LegUp Verilog modules. The output signals for this interface are driven by a priority decoder and multiplexer combination that allows the signals to be sourced from whichever sub-module is currently active in the generated LegUp state machine.

All calls sites to the special functions of "Enqueue", "Dequeue", "Raise", and "Lower" are replaced with the equivalent Twill runtime hardware signaling. Furthermore, all load and store instructions are replaced with the appropriate signaling for interfacing with the Twill runtime hardware memory operations.

Several small modifications were made to how LegUp handles multiplies, division, memory blocks, and PLL blocks in order to use LegUp on Xilinx FPGAs rather than the originally supported Altera FPGAs. Thus, even though Twill has only been tested on Xilinx FPGAs the Twill tool-chain does support programming for Altera based FPGAs.

D. Final Steps

Once the LLVM IR has been broken into standalone parts, the threads designated for the processors are transformed into C with the default LLVM C backend while the hardware threads are transformed into Verilog with LegUp. At this point the Xilinx tools for SOC systems are used to build a bitstream for the FPGA.

V. Results

All of the results presented were measured on a Xilinx XUPV5 board with a Virtex 5 FPGA. The runtime system has also been run on a Nexys 2 board with a Spartan 3E FPGA and a ZedBoard with a Zync-7000 SOC. All of the tests were run with only 8x32 sized queues and with one Microblaze processor. The Microblaze processor is configured to minimize its area according to the Xilinx tools to better simulate a constrained embedded system. All hardware modules including Microblaze are clocked at 100MHz. All HDL code for both LegUp and Twill was synthesized with the "optimize for performance" setting in the Xilinx ISE Project Navigator version 14.6.

The CHStone benchmarks from [11] were used to compare Twill to both the pure software solution and the pure hardware solution. These benchmarks are relatively parallelizable and also are fully supported by LegUp so a baseline could be established. Note that DFAdd, DFDiv, DFMul, and DFSine CHStone benchmarks all utilize 64-bit values and thus were not included since Twill currently does not support larger than 32-bit values.

A. Twill DSWP Results

A summary of the number of hardware threads, queues and semaphores created can be found in Table 6. Across all of the benchmarks, the partitioner generated a workload split of about 75%-25% between the hardware threads and the software thread. The MIPS benchmark and SHA benchmarks both had all of their functions inlined and thus had no function calls to generate new threads. In contrast, the Blowfish benchmark had the largest number of functions that couldn't be extracted into their own thread due to the nature of its optimized call graph.

B. Area Analysis

The runtime system is quite small, using on average across all of the tests 2-4% of the FPGA. Each HWInterface module takes up 44 Look Up Tables (LUTs). An 8x32 queue uses 65 LUTs and one DSP block. Semaphores take up 70 LUTs with 100 primitives on the bus. The processor interface takes up 24 LUTs. The scheduler takes up 98 LUTs and two DSP blocks. Each of the two bus arbiters utilize 15 LUTs apiece.

Table 7 shows the total number of FPGA blocks used by Twill compared against the same benchmark purely translated by LegUp. The Twill HWThreads column consists of only the number of LUTs that the LegUp translated HW threads take up. The Twill column includes the LUTs that the HW threads use along with the runtime system queues, semaphores, busses, and memory cache update system. Finally, the Twill + Microblaze column includes everything from the prior columns along with the LUTs used for the Microblaze soft processor. As can be seen the pure hardware size is always smaller than LegUp's translation mainly due to less functionality existing in the hardware. Adding in the overhead of the runtime system puts Twill's size on par with LegUp's results which is reasonable particularly if a hard processor is being used rather than a soft one. On average, we see a modest 1.73 times area decrease in the space required by the HW Threads and a

Benchmark	# Queues	# Semaphores	#HWThreads
MIPS	12	0	1
ADPCM	328	0	5
AES	100	0	3
Blowfish	104	2	2
GSM	65	0	3
JPEG	576	3	6
MPEG-2	47	0	4
SHA	82	0	1

Fig. 6: DSWP Results

Benchmark	LegUp	Twill HWThreads	Twill	Twill + Microblaze
MIPS	2101	1830	2318	3752
ADPCM	16893	7182	28682	30116
AES	16488	8302	15338	16772
Blowfish	5872	3293	10493	11927
GSM	7397	5888	11983	13417
JPEG	31084	18443	56101	57535
MPEG-2	16295	8116	13467	14901
SHA	12956	7856	13352	14768

Fig. 7: Number of LUTs used in FPGA logic for pure HW translation by LegUp and hybrid Twill implementation

slight increase of 1.35 area increase when including the Twill runtime system.

Aside from LUTs, LegUp makes use of BRAM memory blocks to pass arguments to functions and to handle arrays. Very few BRAM blocks are used in Twill's HW threads while most benchmarks used 10-15 BRAM blocks with the pure LegUp synthesis. Microblaze uses 16 BRAM blocks regardless of what code is running which provides 32kB of instruction and data memory for the Microblaze processor. In addition, with the way that Twill's memory management works almost all of the HW thread data is stored in the processor's data memory segment instead of creating new blocks. This gives all benchmarks comparable numbers of BRAM blocks between LegUp's pure HW translation and Twill's hybrid translation.

C. Power Analysis

Figure 8 shows the power characteristics obtained through Xilinx's power simulation tools. Twill is compared to LegUp's pure HW translation normalized to the pure software implementation running on Microblaze. As expected, the pure HW translation has the best power performance followed by Twill and then the pure Microblaze implementation. This is because Microblaze is really power inefficient compared to a direct hardware implementation. With a hard processor it could be expected that Twill's power consumption would be less than LegUp's since it has to synthesize less hardware. On examining why Microblaze is so inefficient it appears that the majority of the power consumption comes from the multiple Phase-Lock Loops (PLLs) used internally.

D. Performance Analysis

Figure 9 shows the performance characteristics of Twill compared to LegUp's pure HW translation normalized against running the benchmark directly on the Microblaze processor. In general Twill outperforms the pure hardware implementation since it can take advantage of TLP as well as ILP. Twill on average achieves a 1.63 times speedup over the pure hardware implementation on these benchmarks which are designed to be easily translatable into pure hardware. Twill also vastly outperforms a pure SW implementation on the Microblaze processor as expected by on average 22.2 times. This speedup comes from multiple sources: arithmetic operations such as multiply and divide are much faster in hardware, LegUp will schedule as many instructions as possible at the same time to exploit ILP, and Twill will run instructions on the processor at the same time as LegUp is executing its state machine in order to exploit TLP.

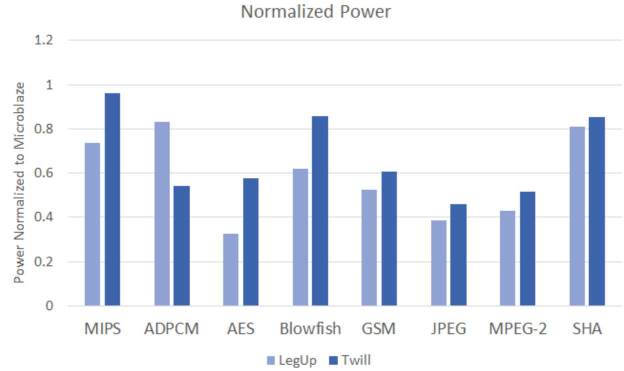


Fig. 8: Power Consumption normalized to the pure Microblaze SW implementation measured using Xilinx's power simulation tools

Twill manages to only match the pure hardware speedup on the Blowfish benchmark. On closer inspection, it appears that Twill chose poor partitions for the hardware and software threads with each function call in the main loop transferring the master control between the hardware and software. This causes the function argument data to be sent back and forth several times between the hardware and software threads before any computation on the data is performed. Similarly, the return value alternates back and forth before finally being used in the next iteration of the loop. We modified the heuristic specifically for this benchmark to prevent this behavior and found a 1.89 times speedup between the modified Twill implementation and the pure hardware implementation. This modified heuristic also decreased the number of queues from 92 to 34 which shows that our original heuristic for partitioning instructions into separate threads could use some improvement.

LegUp appears to do a poor job at synthesizing the ADPCM benchmark compared to the other benchmarks. This interpretation is consistent among the area, power, and performance results. Some of the constructs in this program appear to be quite difficult to synthesize which gives an advantage to Twill when it puts these parts on the processor. This is the only benchmark shown that utilizes division extensively which might be one of the contributing factors since LegUp was set up to use a simple serial divider for these tests.

E. Partitioning Heuristic Effects on Performance

We explored the effects of changing the targeted percentage of instructions to be placed into the partitions. Figures 10 and 11 show the changes in performance and queue count modifying where this split point lies. As can be seen most clearly in Figure 10 there is a negative correlation between the number of queues required and the performance of Twill for a given benchmark. Furthermore, it seems that the even splits between the HW/SW domains perform the worst. This is probably because when the first half of most computations are computed in SW and then the intermediate results are passed to the HW in order to finish the computation the communication costs skyrocket while the amount of TLP exploited remains about the same.

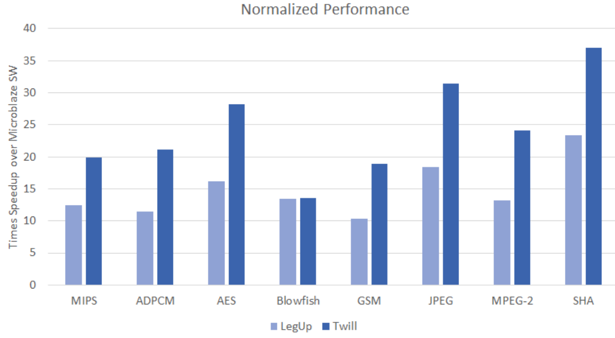


Fig. 9: Performance Speedups normalized to the pure SW implementation

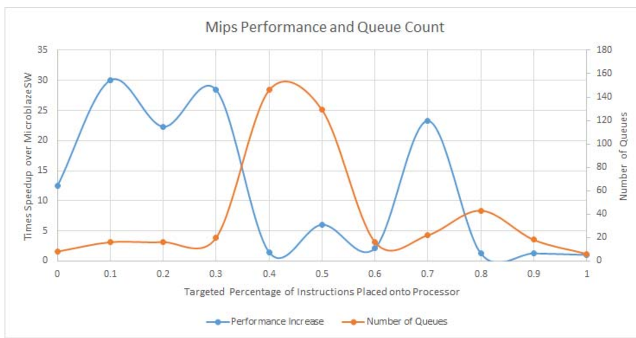


Fig. 10: Mips benchmark performance with various targeted partition split points

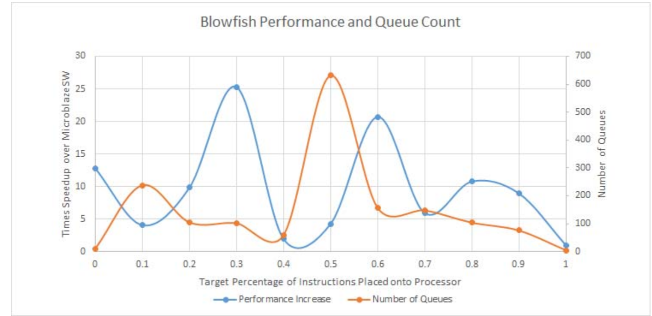


Fig. 11: Blowfish benchmark performance with various targeted partition split points

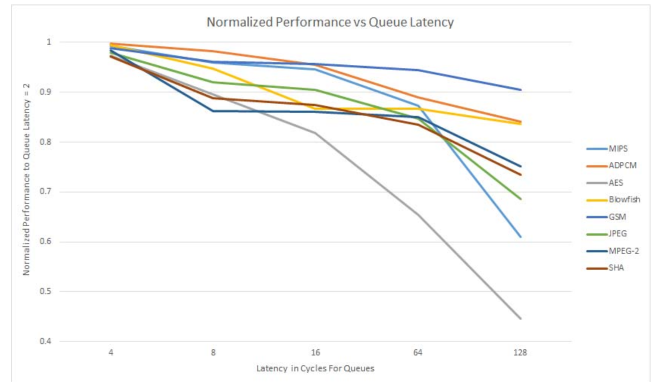


Fig. 12: Twill performance speedups normalized to runtime with 2 cycle queue latency

Ottoni et al. found very similar results when they were experimenting with finding the optimal partitioning for a given loop. While they were very focused on balancing the work across threads in an optimal manner since they assumed homogeneous threads, they found that the greedy heuristic algorithm for partitioning is not particularly good at finding the optimal partition but often works "well enough". That seems to be the case with Twill as well. While perhaps a more complicated heuristic could be used to achieve better results, Twill's results show that its automatic thread extraction through partitioning can result in a significant performance increase without any programmer intervention.

F. Queue Size and Latency

One important result from the DSWP implementation described Ottoni et al. [5] is that the algorithm was very resilient to large queue latencies and short queue sizes regardless of the benchmark run. This was achieved by never having the pipeline "flushed" except for at the very end of program execution. Our implementation of DSWP potentially flushes the pipeline much more frequently on function boundaries and so a similar experiment was conducted to determine the resiliency of Twill to hardware queue latencies and sizes.

Figure 12 shows that while Twill's resiliency depends upon the application, overall Twill is still fairly resilient. Compared to Ottoni et al.'s original implementation of the DSWP algorithm, we have found a much bigger performance degradation as the queue latencies are increased. On average Ottoni et al. report a 10% slowdown with a queue latency of 100 while we found a 27% performance decrease on average with a queue latency of 128. As noted above, this is probably because of how Twill flushes the pipeline fairly frequently. In addition, the original paper only optimized a single long-running loop out of the entire program and thus any performance increase or slowdown effect will be magnified in our full program implementation. Thus we believe that our performance decrease is much closer to the original results than the data suggests.

Figure 13 shows similar results for the queue sizes. Note that for the JPEG benchmark the 32 queue size did not fit on the FPGA. Ottoni et al. found that they received a slowdown of 6% when reducing the queue length from 32 to 8. We found a comparable 9.7% slowdown when comparing our queue lengths of 32 and 8. As mentioned above, our slowdown/speedup results are probably exaggerated compared to the original results; in addition, we used 32-bit queues while the original paper used 8-bit queues.

VI. Conclusion

In this paper we presented a new hybrid SOC compiler and corresponding run-time system called Twill. Twill takes advantage of TLP and ILP in order to achieve a performance

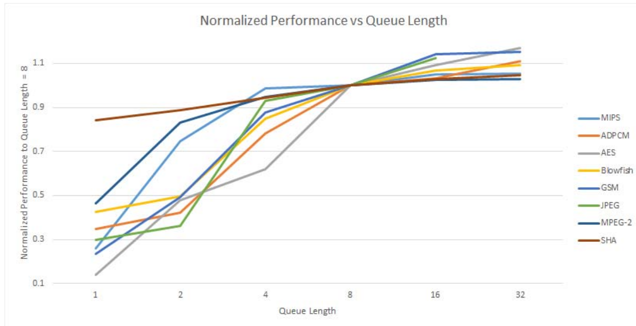


Fig. 13: Twill performance speedups normalized to runtime with length 8 queues

speedup of 1.63 times over LegUp's pure hardware translation even while reducing the amount of area needed for the reconfigurable logic. Twill achieves this by utilizing a modified version of DSWP to extract long-running threads from the input C source and then distributing these threads across the hardware/software divide in a hybrid CPU-FPGA SOC.

A. Future Work

As mentioned in Section II, Twill currently supports only a subset of the C language. Notably, recursion and function pointers are currently not supported. There is no conceptual reason preventing their implementation and we propose several methods to deal with them. Recursion is only a problem in hardware since there is no stack. The Twill DSWP implementation could be extended to support the concept of barriers. At each barrier point all threads would come to the same execution state such that all queues are empty. The recursive function calls represented by backedges in the call graph would then be protected by these barriers on either side with the master function call always being in software. In this way, the recursive functions or chain of functions could be parallelized as normal and then only at the recursion point would the pipeline be flushed and restarted. This would be slower than the equivalent code written as a loop but should still give reasonable speedups over the pure hardware implementation.

A similar system could be used to handle function pointers as well. Everything up to the actual call instruction with the function pointer could be parallelized. Anytime a function pointer is assigned to a new function the code must be changed to assign the master DSWP function. The call could be protected with barriers with the software always having master control of the called function. Furthermore, the way Twill handles function calls would have to change slightly. Instead of having the calling function call all of the slave functions each master DSWP function would be responsible to start the slave functions. This would increase the overhead of function calls slightly but potentially could be limited with points-to analysis to only the functions that could be called through a function pointer.

Another shortcoming of Twill is that it does not support larger than 32 bit data values to be passed inside of queues. This means that 64 bit data types and structures that are bigger than 32 bits are not supported currently by Twill. This shortcoming is relatively easy to overcome; one option is to enqueue/dequeue two or more values at a time and rebuild the resulting data structure or to simply use multiple queues to pass the data.

Another aspect of Twill that can be improved is the partitioning heuristic. As mentioned in Section V-D, the partitioning heuristic can have a huge impact on the final performance of the program. More research is needed into how different heuristics affect this performance and what the best heuristic is for various program types.

Finally, Vachharajani et al. [12] extended the DSWP algorithm to be speculative. This allowed them to greatly increase the speedup gained by the original algorithm with a little hardware support. Since Twill has a large control over the hardware through the reconfigurable logic, it seems relatively straightforward to extend Twill's DSWP algorithm to be speculative which should allow Twill to extract even more long-running threads and increase the amount of TLP parallelization that it can utilize.

References

[1] E. Lubbers and M. Platzner, "ReconOS: An RTOS supporting Hard and Software Threads," in *Field-Programmable Logic and Applications*, 2007, pp. 441–446.

[2] W. Peck, E. Anderson, J. Agron, J. Stevens, F. Baijot, and D. L. Andrews, "Hthreads: A Computational Model for Reconfigurable Devices," in *Field-Programmable Logic and Applications*, 2006, pp. 1–4.

[3] S. Mahadevan, V. S. Gopinath, R. Lysecky, J. Sprinkle, J. W. Rozenblit, and M. W. Marcellin, "Hardware/Software Communication Middleware for Data Adaptable Embedded Systems," in *Engineering of Computer-Based Systems*, 2011.

[4] G. N. thi Huong and S. W. Kim, "Support of cross calls between a microprocessor and FPGA in CPU-FPGA coupling architecture," in *Symposium on Parallel and Distributed Processing*, 2010, pp. 1–8.

[5] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. D. Brown, and T. S. Czajkowski, "LegUp: high-level synthesis for FPGA-based processor/accelerator systems," in *Symposium on Field Programmable Gate Arrays*, 2011, pp. 33–36.

[6] A. Canis, J. Choi, B. Fort, R. Lian, Q. Huang, N. Calagar, M. Gort, J. Qin, M. Aldham, T. Czajkowski, S. Brown, and J. Anderson, "From Software to Accelerators with LegUp High-Level Synthesis," in *Int'l Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2013.

[7] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK: a high-level synthesis framework for applying parallelizing compiler transformations," in *VLSI Design*, 2003, pp. 461–466.

[8] S. S. Huang, A. Hormati, D. F. Bacon, and R. M. Rabbah, "Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary," in *European Conference on Object-Oriented Programming*, 2008, pp. 76–103.

[9] G. Ottoni, R. Rangan, A. Stoler, and D. I. August, "Automatic Thread Extraction with Decoupled Software Pipelining," in *International Symposium on Microarchitecture*, 2005, pp. 105–118.

[10] C. Lattner and V. S. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation," in *Symposium on Code Generation and Optimization*, 2004, pp. 75–88.

[11] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii, "CHStone: A benchmark program suite for practical C-based high-level synthesis," in *IEEE International Symposium on Circuits and Systems*, 2008, pp. 1192–1195.

[12] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August, "Speculative Decoupled Software Pipelining," in *International Conference on Parallel Architectures and Compilation Techniques*, 2007, pp. 49–59.