Hindawi Publishing Corporation
EURASIP Journal on Applied Signal Processing
Volume 2006, Article ID 46472, Pages 1–23
DOI 10.1155/ASP/2006/46472

Rapid VLIW Processor Customization for Signal Processing Applications Using Combinational Hardware Functions

Raymond R. Hoare, Alex K. Jones, Dara Kusic, Joshua Fazekas, John Foster, Shenchih Tung, and Michael McCloud

Department of Electrical and Computer Engineering, University of Pittsburgh, Pittsburgh, PA 15261, USA

Received 12 October 2004; Revised 30 June 2005; Accepted 12 July 2005

This paper presents an architecture that combines VLIW (very long instruction word) processing with the capability to introduce application-specific customized instructions and highly parallel combinational hardware functions for the acceleration of signal processing applications. To support this architecture, a compilation and design automation flow is described for algorithms written in C. The key contributions of this paper are as follows: (1) a 4-way VLIW processor implemented in an FPGA, (2) large speedups through hardware functions, (3) a hardware/software interface with zero overhead, (4) a design methodology for implementing signal processing applications on this architecture, (5) tractable design automation techniques for extracting and synthesizing hardware functions. Several design tradeoffs for the architecture were examined including the number of VLIW functional units and register file size. The architecture was implemented on an Altera Stratix II FPGA. The Stratix II device was selected because it offers a large number of high-speed DSP (digital signal processing) blocks that execute multiply-accumulate operations. Using the MediaBench benchmark suite, we tested our methodology and architecture to accelerate software. Our combined VLIW processor with hardware functions was compared to that of software executing on a RISC processor, specifically the soft core embedded NIOS II processor. For software kernels converted into hardware functions, we show a hardware performance multiplier of up to 230 times that of software with an average 63 times faster. For the entire application in which only a portion of the software is converted to hardware, the performance improvement is as much as 30X times faster than the nonaccelerated application, with a 12X improvement on average.

1. INTRODUCTION

In this paper, we present an architecture and design methodology that allows the rapid creation of application-specific hardware accelerated processors for computationally intensive signal processing and communication codes. The target technology is suitable for field programmable gate arrays (FPGAs) with embedded multipliers and for structured or standard cell application-specific integrated circuits (ASICs). The objective of this work is to increase the performance of the design and to increase the productivity of the designer, thereby enabling faster prototyping and time-to-market solutions with superior performance.

The design process in a signal processing or communications product typically involves a top-down design approach with successively lower level implementations of a set of operations. At the most abstract level, the systems engineer designs the algorithms and control logic to be implemented in a high level programming language such as Matlab or C. This functionality is then rendered into a piece of hardware, either by a direct VLSI implementation, typically on either an FPGA platform or an ASIC, or by porting the system code to a microprocessor or digital signal processor (DSP). In fact, it is very common to perform a mixture of such implementations for a realistically complicated system, with some functionality residing in a processor and some in an ASIC. It is often difficult to determine in advance how this separation should be performed and the process is often wrought with errors, causing expensive extensions to the design cycle.

The computational resources of the current generation of FPGAs and of ASICs exceed that of DSP processors. DSP processors are able to execute up to eight operations per cycle while FPGAs contain tens to hundreds of multiply-accumulate *DSP blocks* implemented in ASIC cells that have configurable width and can execute sophisticated multiply-accumulate functions. For example, one DSP block can execute $A * B \pm C * D + E * F \pm G * H$ in two cycles on

9-bit data or it can execute $A * B + C$ on 36-bit data in two cycles. An Altera Stratix II contains 72 such blocks as well as numerous logic cells [1]. Xilinx has released preliminary information on their largest Virtex 4 that will contain 512 multiply-accumulate ASIC cells, with 18x18-bit multiply and a 42-bit accumulate, and operate at a peak speed of 500 MHz [2]. Lattice Semiconductor has introduced a low-cost FPGA that contains 40 DSP blocks [3]. From our experiments, a floating point multiplier/adder unit can be created using 4 to 8 DSP blocks, depending on the FPGA.

Additionally, ASICs can contain more computational power than an FPGA but consume much less power. In fact, there are many companies, including the FPGA vendors themselves, that will convert an FPGA design into an equivalent ASIC and thereby reduce the unit cost and power consumption.

In spite of these attractive capabilities of FPGA architectures, it is often intractable to implement an entire application in hardware. Computationally complex portions of the applications, or *computational kernels*, with generally high available parallelism are often mapped to these devices while the remaining portion of the code is executed with a sequential processor.

This paper introduces an architecture and a design methodology that combines the computational power of application-specific hardware with the programmability of a software processor.

The architecture utilizes a tightly coupled general-purpose 4-way very long instruction world (VLIW) processor with multiple application-specific hardware functions. The hardware functions can obtain a performance speedup of 10x to over 100x, while the VLIW can achieve a 1x to 4x speedup, depending on the available instruction level parallelism (ILP). To demonstrate the validity of our solution, a 4-way VLIW processor (pNIOS II) was created based on the instruction set of the Altera NIOS II processor. A high-end 90 nm FPGA, an Altera Stratix II, was selected as the target technology for our experiments.

For the design methodology, we assume that the design has been implemented in strongly typed software language, such as C, or utilizes a mechanism that statically indicate the data structure sizes, like vectorized Matlab. The software is first profiled to determine the critical loops within the program that typically consume 90% of the execution time. The control portion of each loop remains in software for execution on the 4-way VLIW processor. Some control flow from loop structures is removed by loop unrolling. By using predication and function inlining, the entire loop body is converted into a single data flow graph (DFG) and synthesized into an entirely combinational *hardware function*. If the loop does not yield a sufficiently large DFG, the loop is considered for unrolling to increase the size of the DFG. The hardware functions are tightly integrated into the software processor through a shared register file so that, unlike a bus, there is no hardware/software interface overhead. The hardware functions are mapped into the processor's instruction stream as if they are regular instructions except that they require multiple cycles to compute. The exact timing of the hardware functions is determined by the synthesis tool using static timing analysis.

In order to demonstrate the utility of our proposed design methodology, we consider several representative problems that arise in the design of signal processing systems in detail. Representative problems are chosen in the areas of (1) voice compression with the G.721, GSM 06.10, and the proposed CCIIT ADPCM standards; (2) image coding through the inverse discrete cosine transform (IDCT) that arise in MPEG video compression; and (3) multiple-input multiple-output (MIMO) communication systems through the sphere decoder [4] employing the Fincke-Pohst algorithm [5].

The key contributions of this work are as follows.

(i) A complete *32-bit 4-way VLIW soft core processor in an FPGA*. Our pNIOS II processor has been tested on a Stratix II FPGA device and runs at 166 MHz.

(ii) *Speedups* over conventional approaches through hardware kernel extraction and custom implementation in the same FPGA device.

(iii) A *hardware/software interface requiring zero cycle overhead*. By allowing our hardware functions direct access to the entire register file, the hardware function can operate without the overhead of a bus or other bottlenecks. We show that the additional hardware cost to achieve this is minimal.

(iv) A *design methodology* that allows standard applications written in C to map to our processor using a VLIW compiler that automatically extracts available parallelism.

(v) *Tractable design automation techniques* for mapping computational kernels into efficient custom combinational hardware functions.

The remainder of the paper is organized as follows: we provide some motivation for our approach and its need in signal processing in Section 2. In Section 3, we describe the related work to our architecture and design flow. Our architecture is described in detail in Section 4. Section 5 describes our design methodology including our method for extracting and synthesizing hardware functions. Our signal processing applications are presented in Section 6 including an in depth discussion of our design automation techniques using these applications as examples. We present performance results of our architecture and tool flow in Section 7. Finally, Section 8 describes our conclusions with planned future work.

2. MOTIVATION

The use of FPGA and ASIC devices is a popular method for speeding up time critical signal processing applications. FPGA/ASIC technologies have seen several key advancements that have led to greater opportunity for mapping these applications to FPGA devices. ASIC cells such as DSP blocks and block RAMs within FPGAs provide an efficient method to supplement increasing amounts of programmable logic within the device. This trend continues to increase the complexity of applications that may be implemented and the achievable performance of the hardware implementation.

However, signal processing scientists work with software systems to implement and test their algorithms. In general, these applications are written in C and more commonly in Matlab. Thus, to supplement the rich amount of hardware logic in FPGAs, vendors such as Xilinx and Altera have released both FPGAs containing ASIC processor cores such as the PowerPC enabled Virtex II Pro and the ARM-enabled Excalibur, respectively. Additionally, Xilinx and Altera also produce soft core processors Microblaze and NIOS, each of which can be synthesized on their respective FPGAs.

Unfortunately, these architectures have several deficiencies that make them insufficient alone. Hardware logic is difficult to program and requires hardware engineers who understand the RTL synthesis tools, their flow, and how to design algorithms using cumbersome hardware description languages (HDLs). Soft core processors have the advantage of being customizable making it easy to integrate software and hardware solutions in the same device. However, these processors are also at the mercy of the synthesis tools and often cannot achieve necessary speeds to execute the software portions of the applications efficiently. ASIC core processors provide much higher clock speeds; however, these processors are not customizable and generally only provide bus-based interfaces to the remaining FPGA device creating a large data transfer bottleneck.

Figure 1 displays application profiling results for the SpecInt, MediaBench, and NetBench suites, with a group of selected security applications [5]. The 90/10 rule tells us that on average, 90% of the execution time for an application is contained within about 10% of the overall application code. These numbers are an average of individual application profiles to illustrate the overall tendency of the behavior of each suite of benchmarks. As seen in Figure 1, it is clear that the 10% of code referred to in the 90/10 rule refers to loop structures in the benchmarks. It is also apparent that multimedia, networking, and security applications, this includes several signal processing benchmark applications, exhibit even higher propensity for looping structures to make a large impact on the total execution time of the application.

Architectures that take advantage of parallel computation techniques have been explored as a means to support computational density for the complex operations required by digital processing of signals and multimedia data. For example, many processors contain SIMD (single instruction multiple data) functional units for vector operations often found in DSP and multimedia codes.

VLIW processing improves upon the SIMD technique by allowing each processing element parallelism to execute its instructions. VLIW processing alone is still insufficient to achieve significant performance improvements over sequential embedded processing. When one considers a traditional processing model that requires a cycle for operand-fetch, execute, and writeback, there is significant overhead that occupies what could otherwise be computation time. While pipelining typically hides much of this latency, misprediction of branching reduces the processor ILP. A typical software-level operation can take tens of instructions more than the alternative of a single, hardware-level operation that propagates the results from one functional unit to the next without the need for write-back, fetch, or performance-affecting data forwarding.

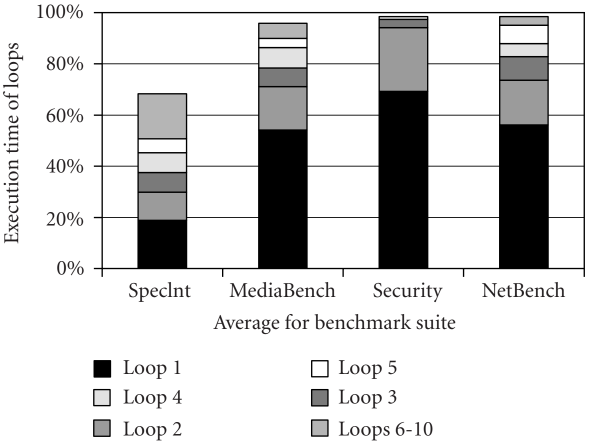


Figure 1: Execution time contained within the top 10 loops in the code averaged across the SpecInt, MediaBench, and NetBench suites, as well as selected security applications [5].

Our technique for extracting computational kernels in the form of loops from the original code for no overhead implementation in combinational hardware functions allows the opportunity for large speedups over traditional or VLIW processing alone. We have mapped a course-grain computational structure on top of the fine-grain FPGA fabric for implementation of hardware functions. In particular, this hardware fabric is coarse-grained and takes advantage of extremely low-latency DSP (multiply-accumulate) blocks implemented directly in silicon. Because the fabric is combinational, no overhead from nonuniform or slow datapath stages is introduced.

For implementation, we selected an Altera Stratix II EP2S180F1508C4 in part for its high density of sophisticated DSP multiply-accumulate blocks and the FPGA's rapidly maturing tool flow that eventually permits fine grain control over routing layouts of the critical paths. The FPGA is useful beyond prototyping, capably supporting deployment with a maximum internal clock speed of 420 MHz dependent on the interconnect of the design and on-chip resource utilization. For purposes of comparing performance, we compare our FPGA implementation against our implementation of the Altera NIOS II soft core processor.

3. RELATED WORK

Manual hardware acceleration has been applied to countless algorithms and is beyond enumeration here. These systems generally achieve significant speedups over their software counterparts. Behavioral and high-level synthesis techniques attempt to leverage hardware performance from different levels of behavioral algorithmic descriptions. These different representations can be from hardware description languages (HDLs) or software languages such as C, C++, Java, and Matlab.

The HardwareC language is a C-like HDL used by the Olympus synthesis system at Stanford [6]. This system uses high-level synthesis to translate algorithms written in HardwareC into standard cell ASIC netlists. Esterel-C is a system-level synthesis language that combines C with the Esterel language for specifying concurrency, waiting, and pre-emption developed at Cadence Berkeley Laboratories [7]. The SPARK synthesis engine from the UC Irvine translates algorithms written in C into hardware descriptions emphasizing extraction of parallelism in the synthesis flow [8, 9]. The PACT behavioral synthesis tool from Northwestern University translates algorithms written in C into synthesizable hardware descriptions that are optimized for low-power as well as performance [10, 11].

In industry, several tools exist which are based on behavioral synthesis. The Behavioral Compiler from Synopsys translates applications written in SystemC into netlists targeting standard cell ASIC implementations [12, 13]. SystemC is a set of libraries designed to provide HDL-like functionality within the C++ language for system level synthesis [14]. Synopsys cancelled its Behavioral Compiler because customers were unwilling to accept reduced quality of results compared to traditional RTL synthesis [15]. Forte Design Systems has developed the Cynthesizer behavioral synthesis tool that translates hardware independent algorithm descriptions in C and C++ into synthesizable hardware descriptions [16]. Handel-C is a C-like design language from Celoxica for system level synthesis and hardware software co-design [17]. Accelchip provides the AccelFPGA product, which translates Matlab programs into synthesizable VHDL for synthesis on FPGAs [18]. This technology is based on the MATCH project at Northwestern [19]. Catapult C from Mentor Graphics Corporation translates a subset of untimed C++ directly into hardware [20].

The difference between these projects and our technique is that they try to solve the entire behavioral synthesis problem. Our approach utilizes a 4-wide VLIW processor to execute nonkernel portions of the code (10% of the execution time) and utilizes tightly coupled hardware acceleration using behavioral synthesis of kernel portions of the code (90% of the execution time). We match the available hardware resources to the impact on the application performance so that our processor core utilizes 10% or less of the hardware resources leaving 90% or more to improve the performance of the kernels.

Our synthesis flow utilizes a DFG representation that includes *hardware predication*: a technique to convert control flow based on conditionals into multiplexer units that select from two inputs from this conditional. This technique is similar to assignment decision diagram (ADD) representation [21, 22], a technique to represent functional register transfer level (RTL) circuits as an alternative to control and data flow graphs (CDFGs). ADDs read from a set of primary inputs (generally registers) and compute a set of logic functions. A conditional called an *assignment decision* then selects an appropriate output for storage into internal storage elements. ADDs are most commonly used for automated generation of test patterns for circuit verification [23, 24]. Our technique is not limited to decisions saved to internal storage, which imply sequential circuits. Rather, our technique applies *hardware predication* at several levels within a *combinational* (i.e., DFG) representation.

The support of custom instructions for interface with coprocessor arrays and CPU peripherals has developed into a standard feature of soft-core processors and those which are designed for DSP and multimedia applications. Coprocessor arrays have been studied for their impact on speech coders [25, 26], video encoders [27, 28], and general vector-based signal processing [29–31].

These coprocessor systems often assume the presence and interface to a general-purpose processor such as a bus. Additionally, processors that support custom instructions for interface to coprocessor arrays are often soft-core and run a significantly slower clock rates than hard-core processors. Our processor is fully deployed on an FPGA system with detailed post place-and-route performance characterization. Our processor does not have the performance bottleneck associated with a bus interconnect but directly connects the hardware unit to the register file. There is no additional overhead associated with calling a hardware function.

Several projects have experimented with reconfigurable functional units for hardware acceleration. PipeRench [32–36] and more recently HASTE [37] have explored implementing computational kernels on coarse-grained reconfigurable fabrics for hardware acceleration. PipeRench utilizes a pipeline of subword ALUs that are combined to form 32-bit operations. The limitation of this approach is the requirement of pipelining as more complex operations require multiple stages and, thus, incur latency. In contrast, we are using non-clocked hardware functions that represent numerous 32-bit operations. RaPid [38–42] is a coarse-grain reconfigurable datapath for hardware acceleration. RaPid is a datapath-based approach and also requires pipelining. Matrix [43] is a coarse-grained architecture with an FPGA like interconnect. Most FPGAs offer this coarse-grain support with embedded multipliers/adders. Our approach, in contrast, reduces the execution latency and, thus, increases the throughput of computational kernels.

Several projects have attempted to combine a reconfigurable functional unit with a processor. The Imagine processor [44–46] combines a very wide SIMD/VLIW processor engine with a host processor. Unfortunately, it is difficult to achieve efficient parallelism through high ILP due to many types of dependencies. Our processor architecture differs as it uses a flexible combinational hardware flow for kernel acceleration.

The Garp processor [47–49] combines a custom reconfigurable hardware block with a MIPS processor. In Garp, the hardware unit has a special purpose connection to the processor and direct access to the memory. The Chimaera processor [50, 51] combines a reconfigurable functional unit with a register file with a limited number of read and write ports. Our system differs as we use a VLIW processor instead of a single processor and our hardware unit connects directly to all registers in the register file for both reading and writing allowing hardware execution with no overhead. These projects also assume that the hardware resource must be reconfigured to execute a hardware-accelerated kernel, which may require significant overhead. In contrast, our system configures the hardware blocks prior to runtime and uses multiplexers to select between them at runtime. Additionally, our system is physically implemented in a single FPGA device, while it appears that Garp and Chimaera were studied in simulation only.

In previous work, we created a 64-way and an 88-way SIMD architecture and interconnected the processing elements (i.e., the ALUs) using a hypercube network [52]. This architecture was shown to have a modest degradation in performance as the number of processors scaled from 2 to 88. The instruction broadcasting and the communication routing delay were the only components that degraded the scalability of the architecture. The ALUs were built using embedded ASIC multiply-add circuits and were extended to include user-definable instructions that were implemented in FPGA gates. However, one limitation of a SIMD architecture is the requirement for regular instructions that can be executed in parallel, which is not the case for many signal processing applications. Additionally, explicit communications operations are necessary.

Work by industry researchers [53] shows that coupling a VLIW with a reconfigurable resource offers the robustness of a parallel, general-purpose processor with the accelerating power and flexibility of a reprogrammable systolic grid. For purposes of extrapolation, the cited research assumes the reconfiguration penalty of the grid to be zero and that design automation tools tackle the problem of reconfiguration. Our system differs because the FPGA resource can be programmed prior to execution, giving us a more realistic reconfiguration penalty of zero. We also provide a compiler and automation flow to map kernels onto the reconfigurable device.

4. ARCHITECTURE

The architecture we are introducing is motivated by four factors: (1) the need to accelerate applications within a single chip, (2) the need to handle real applications consisting of thousands of lines of C source code, (3) the need to achieve speedup when parallelism does not appear to be available, and (4) the size of FPGA resources continues to grow as does the complexity of fully utilizing these resources.

Given these needs, we have created a VLIW processor from the ground-up and optimized its implementation to utilize the DSP Blocks within an FPGA. A RISC instruction set from a commercial processor was selected to validate the completeness of our design and to provide a method of determining the efficiency of our implementation.

In order to achieve custom hardware speeds, we enable the integration of hardware and software within the same processor architecture. Rather than adding a customized coprocessor to the processor's I/O bus that must be addressed through a memory addressing scheme, we integrated the execution of the hardware blocks as if it was a custom instruction. However, we have termed the hardware blocks as *hardware functions* because they perform the work of tens to hundreds of assembly instructions. To eliminate data movement, our hardware functions share the register file with the processor and, thus, the overhead involved in calling a hardware function is exactly that of an inlined software functions.

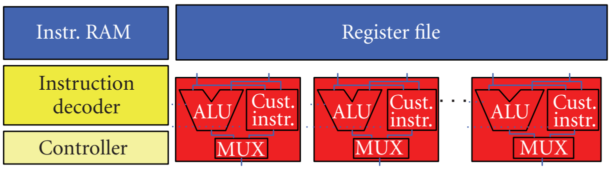


Figure 2: Very long instruction word architecture.

These hardware functions can be multiple cycles and are scheduled as if it were just another software instruction. The hardware functions are purely combinational (i.e., not internally registered) and receive their data inputs from the register file and return computed data to the register file. They contain predication operations and are the hardware equivalent of tens to hundreds of assembly instructions. These features enable large speedup with zero-overhead hardware/software switching. The following three subsections describe each of the architectural components in detail.

From Amdahl's Law of speedup, we know that even if we infinitely speedup 90% of the execution time, we will have a maximum of 10X speedup if we ignore the remaining 10% of the time. Thus, we have taken a VLIW architecture as the baseline processor and sought to increase its width as much as possible within an FPGA. An in-depth analysis and performance results show the limited scalability of a VLIW processor within an FPGA.

4.1. *VLIW processor*

To ensure that we are able to compile any C software codes, we implemented a sequential processor based on the NIOS II instruction set. Thus, our processor, pNIOS II, is binary-code-compatible to the Altera NIOS II soft-core processor. The branch prediction unit and the register windowing of the Altera NIOS II have not been implemented at the time of this publication.

In order to expand the problem domains that can be improved by parallel processing within a chip, we examined the scalability of a VLIW architecture for FPGAs. As shown in Figure 2, the key differences between VLIWs and SIMDs or MIMDs are the wider instruction stream and the shared register file, respectively. The ALUs (also called PEs) can be identical to that of their SIMD counterpart. Rather than having a single instruction executed each clock cycle, a VLIW can execute P operations for a P processor VLIW.

We designed and implemented a 32-bit, 6-stage pipelined soft-core processor that supports the full NIOS II instruction set including custom instructions. The single processor was

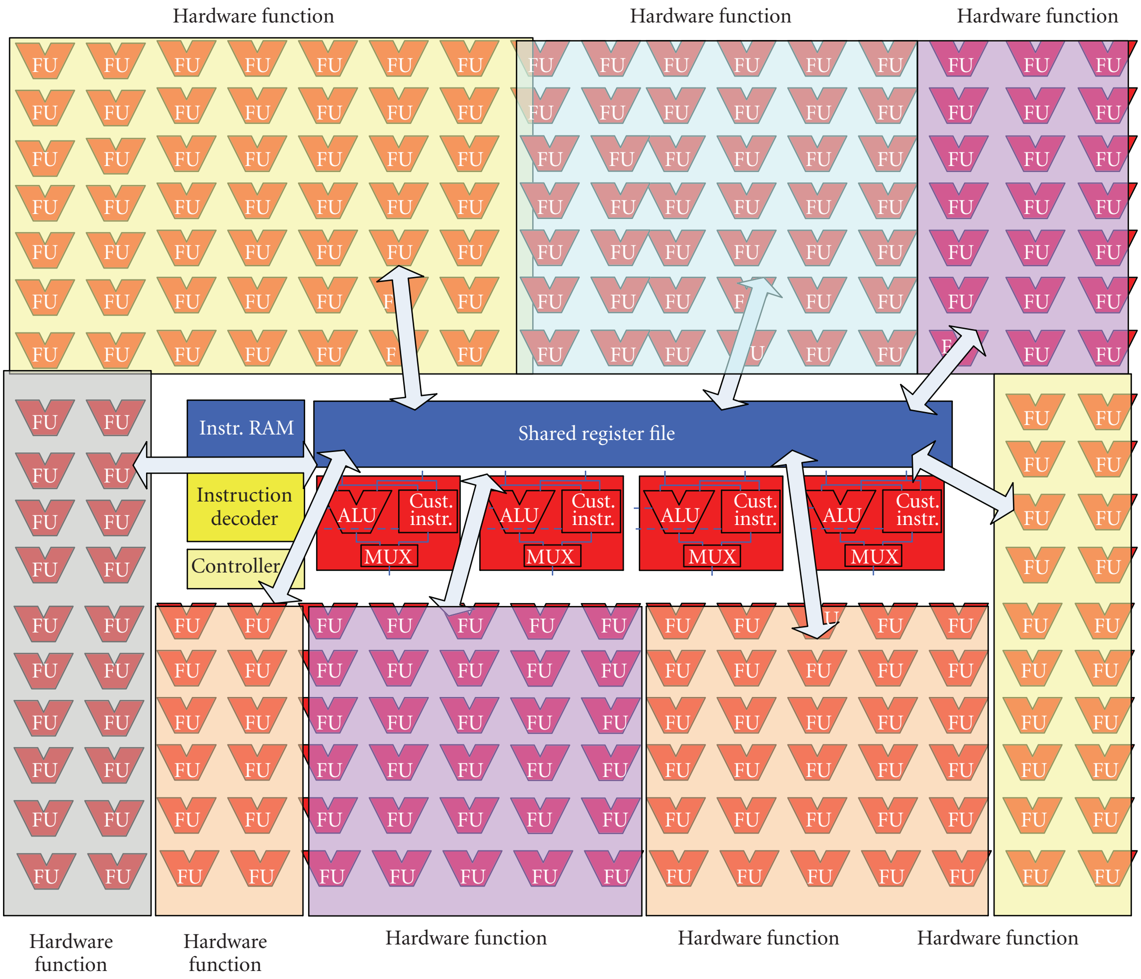


Figure 3: The VLIW processor architecture with application-specific hardware functions.

then configured in a 4-wide VLIW processor using a shared register file. The shared 32-element register file has 8 read ports and 4 write ports.

There is also a 16 KB dual-ported memory accessible to 2 processing elements (PEs) in the VLIW, and a single 128-bit wide instruction ROM. An interface controller arbitrates between software and hardware functions as directed by the custom instructions.

We targeted our design to the Altera Stratix II EP2-S180F1508C4 FPGA with a maximum internal clock rate of 420 MHz. The EP2S180F has 768 9-bit embedded DSP multiply-adders and 1.2 MB of available memory. The single processor was iteratively optimized to the device based on modifications to the critical path. The clock rate sustained increases to its present 4-wide VLIW rate of 166 MHz.

4.2. *Zero-cycle overhead hardware/software interface*

In addition to interconnecting the VLIW processors, the register file is also available to the hardware functions, as shown by an overview of the processor architecture in Figure 3 and through a register file schematic in Figure 4. By enabling the compiler to schedule the hardware functions as if they were software instructions, there is no need to provide an additional hardware interface. The register file acts as the data buffer as it normally does for software instructions. Thus, when hardware function needs to be called, its parameters are stored in the register file for use by the hardware function. Likewise, the return value of the hardware function is placed back into the register file.

The gains offered by a robust VLIW supporting a large instruction set come at a price to the performance and area of the design. The number of ports to the shared register file and instruction decode logic have shown in our tests to be the greatest limitations to VLIW scalability. A variable-sized register file is shown in.

In Figure 4, P processing elements interface to N registers. Multiplexing breadth and width pose the greatest hindrances to clock speed in a VLIW architecture. We tested the effect of multiplexers by charting performance impact by increasing the number of ports on a shared register file, an expression of increasing VLIW width.

In Figure 5, the number of 32-bit registers is fixed to 32 and the number of processors is scaled. For each processor, two operands need to be read and one written per cycle. Thus, for P processors there are $2P$ read ports and

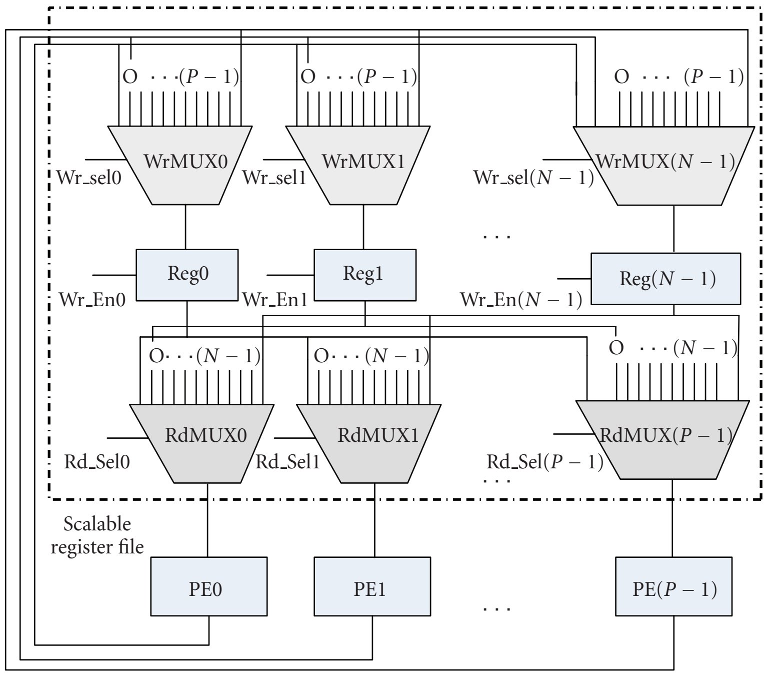


Figure 4: N-element register file supporting P-wide VLIW with P read ports and P write ports.

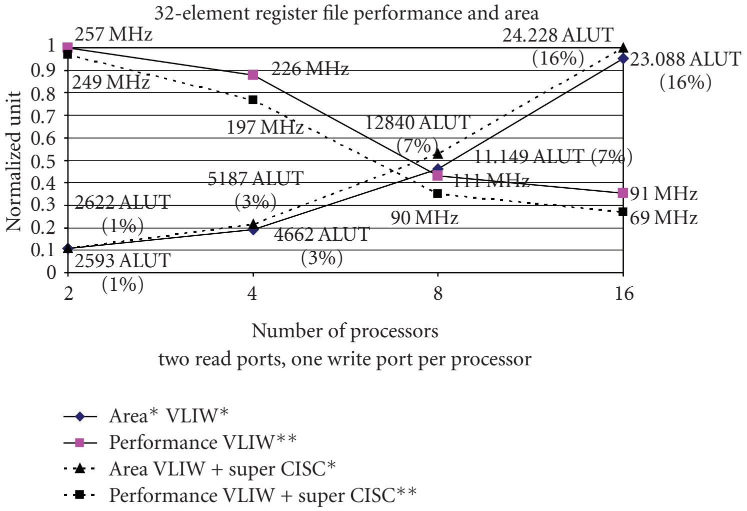


Figure 5: Scalability of a 32-element register file for P processors having $2P$ read and P write ports. Solid lines are for just a VLIW while dashed lines include access for SuperCISC hardware functions. (*Area normalized as percentage of area of 16 processor register file; **performance normalized as percentage of performance of 2 processor register file.)

P write ports. As shown, the performance steadily drops and the number of processors is increased. Additionally, the routing resources and logic resources required also increases.

From an analysis of the benchmarks we examined, we found an average ILP between 1 and 2 and concluded that a 4-way VLIW was more than sufficient for the 90% of the code that requires 10% of the time. We also determined that critical path within the ALU was limited to 166 MHz as seen in Table 1. The performance is limited by the ALU and not the register file. Scaling to 8 or 16-way VLIW would decrease the clock rate of the design, as shown in Figure 5.

The multiplexer is the design unit that contributes most to performance degradation of the register file as the VLIW scales. We measured the impact of a single 32-bit P-to-1 multiplexer on the Stratix II EP2S180. As the width P doubled, the area increased by a factor of 1.4x times the width. The performance took the greatest hit of all our scaling tests, losing an average of 44 MHz per doubling, as shown in Figure 6. The performance degrades because the number of P-to-1

Table 1: Performance of instructions (Altera Stratix II FPGA EP2S180F1508C4).

Post-place and route results for ALU modules on EP2S180F1508C4				
	ALUTs	% Area	Clock	Latency
Adder/subtractor/comparator	96	< 1	241 MHz	4 ns
32-bit integer multiplier (1 cycle)	0 + 8 DSP units	< 1	322 MHz	3 ns
Logical unit (AND/OR/XOR)	96	< 1	422 MHz	2 ns
Variable left/right shifter	135	< 1	288 MHz	4 ns
Top ALU (4 modules above)	416+ DSP units	< 1	166 MHz	6 ns

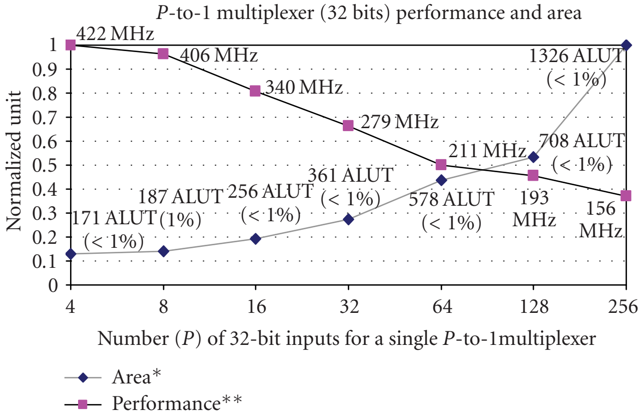


Figure 6: Scalability of a 32-bit *P*-to-1 multiplexer on an Altera Stratix II (EP2S180F1508C4). (*Area normalized as percentage of 256-to-1 multiplexer area; **performance normalized as percentage of 4-to-1 multiplexer performance.)

multiplexers increases to implement the read and write ports within the register file.

For an N-wide VLIW, the limiting factor will be the register file which in turn requires $2N$ $R : 1$ multiplexer as each processor reads two registers from a register file with R registers. For the write ports, each of the R registers requires an a $N : 1$ multiplexer. However, as shown in Figure 5, the logic required for a 4-wide VLIW with 32 shared registers of 32-bits each, only achieved 226 MHz while the 32 : 1 multiplexer achieved 279 MHz. What is not shown is the routing. These performance numbers should be taken as minimums and maximums for the performance of the register file. We were able to scale our VLIW with 32 shared registers up to 166 MHz 4-way.

One technique for increasing the performance of shared register files for VLIW machines is partitioned register files [54]. This technique partitions the original register file into banks of limited connectivity register files that are accessible by a subset of the VLIW processing elements. Busses are used to interconnect these partitions. For a register to be accessed by a processing element outside of the local partition, the data must be moved over a bus using an explicit move instruction. While we considered this technique, we did not employ register file partitioning in our processing scheme for several reasons: (1) the amount of ILP available from our VLIW compiler was too low to warrant more than a 4-way VLIW, (2) the nonpartitioned register file approach was not the limiting factor for performance in our 4-way VLIW implementation, and (3) our VLIW compiler does not support partitioned register files.

4.3. Achieving speedup through hardware functions

By using multicycle hardware functions, we are able to place hundreds of machine instructions into a single hardware function. This hardware function is then converted into logic and synthesized into hardware. The architecture interfaces an arbitrary number of hardware functions to the register file while the compiler schedules the hardware functions as if they were software.

Synchronous design is by definition inefficient. The entire circuit must execute at the rate of the slowest component. For a processor, this means that a simple left-shift requires as much time as a multiply. For kernel codes, this effect is magnified.

As a point of reference, we have synthesized various arithmetic operations for a Stratix II FPGA. The objective is not the absolute speed of the operations but the *relative* speed. Note that a logic operation can execute 5x faster than the entire ALU. Thus, by moving data flow graphs directly into hardware, the critical path from input to output is going to achieve large speedup. The critical path through a circuit is unlikely to contain only multipliers and is expected to be a variety of operations and, thus, will have a smaller delay than if they were executed on a sequential processor.

This methodology requires a moderate sized data flow diagram. There are numerous methods for achieving this and will be discussed again in the following section. One method that requires hardware support is the predication operation. This operation is a conditional assignment of one register to another based on whether the contents of a third register is a "1." This simple operation enables the removal of jumps for if-then-else statements. In compiler terms, predication enables the creation of large data flow diagrams that exceed the size of basic blocks.

5. COMPILATION FOR THE VLIW PROCESSOR WITH HARDWARE FUNCTIONS

Our VLIW processor with hardware functions is designed to assist in creating a *tractable* synthesis tool flow which is outlined in Figure 7. First, the algorithm is profiled using the

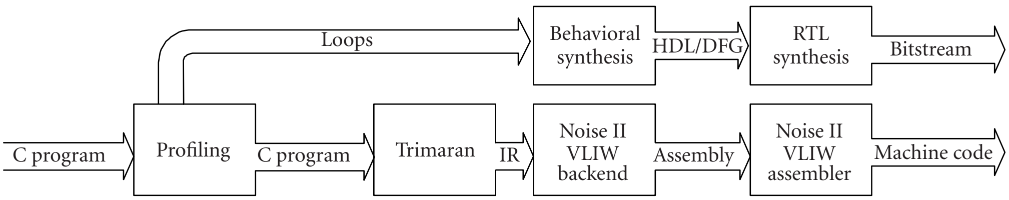


Figure 7: Tool flow for the VLIW processor with hardware functions.

Shark profiling tool from Apple Computer [4] that can profile programs compiled with the gcc compiler. Shark is designed to identify the computationally intensive loops.

The computational kernels discovered by Shark are propagated to a synthesis flow that consists of two basic stages. First, a set of well-understood compiler transformations including function inlining, loop unrolling, and code motion are used to attempt to segregate the loop control and memory accesses from the computation portion of the kernel code. The loop control and memory accesses are sent to the software flow while the computational portion is converted into hardware functions using a behavioral synthesis flow.

The behavior synthesis flow converts the computational kernel code into a CDFG representation. We use a technique called *hardware predication* to merge basic blocks in the CDFG to create a single, larger DFG. This DFG is directly translated into equivalent VHDL code and synthesized for the Stratix II FPGA. Because control flow dependencies between basic blocks are converted into data dependencies using hardware predication, the result is an entirely combinational hardware block.

The remainder of the code, including the loop control and memory access portions of the computational kernels, is passed through the Trimaran VLIW Compiler [55] for execution on the VLIW processor core. Trimaran was extended to generate assembly for a VLIW version of the NIOS II instruction set architecture. This code is assembled by our own assembler into machine code that directly executes on our processor architecture. Details on the VLIW NIOS II backend and assembler are available in [56].

5.1. Performance code profiling

The Shark profiling tool is designed to discover the loops that contribute the most to the total program execution time. The tool returns results such as those seen in Algorithm 2. These are the top two loops from the G.721 MediaBench benchmark that total nearly 70% of the total program execution time.

After profiling, the C program is modified to include directives within the code to signal which portions of the code had been detected to be computational kernels during the profiling. As seen in Algorithm 1, the computational kernel portions are enclosed with the #pragma HW_START and #pragma HW_END directives to denote the beginning and ending of the kernel, respectively. The compiler uses these directives to identify the segments of code to implement in custom hardware.

```
predictor_zero()
0.80% for (i = 1; i < 6; i++) /* ACCUM */
34.60     sezi += fmult (state_ptr−>b[i] >> 2,
              state_ptr−>dq[i]);
- -
35.40%

quan()
14.20% for (i = 0; i < size; i++)
18.10% if (val < *table++)
1.80% break;
- -
33.60%
```

Algorithm 1: Excerpt of profiling results for the G.721 benchmark.

```
predictor_zero()
#pragma HW_START
for (i = 1; i < 6; i++) /* ACCUM */
sezi += fmult(state_ptr−>b[i] >> 2,
                        state_ptr−>dq[i]);
#pragma HW_END

quan()
#pragma HW_START
for (i = 0; i < size; i++)
  if (val < *table++)
    break;
#pragma HW_END
```

Algorithm 2: Code excerpt from Algorithm 1 after insertion of directives to outline computational kernels that are candidates for custom hardware implementation.

5.2. Compiler transformations for synthesis

Synthesis from behavioral descriptions is an active area of study with many projects that generate hardware descriptions from a variety of high-level languages and other behavioral descriptions, see Section 3. However, synthesis of *combinational* logic from properly formed behavioral descriptions is significantly more mature than the general case and can produce efficient implementations. Combinational logic, by definition, does not contain any timing or storage constraints but defines the output as purely a function of the

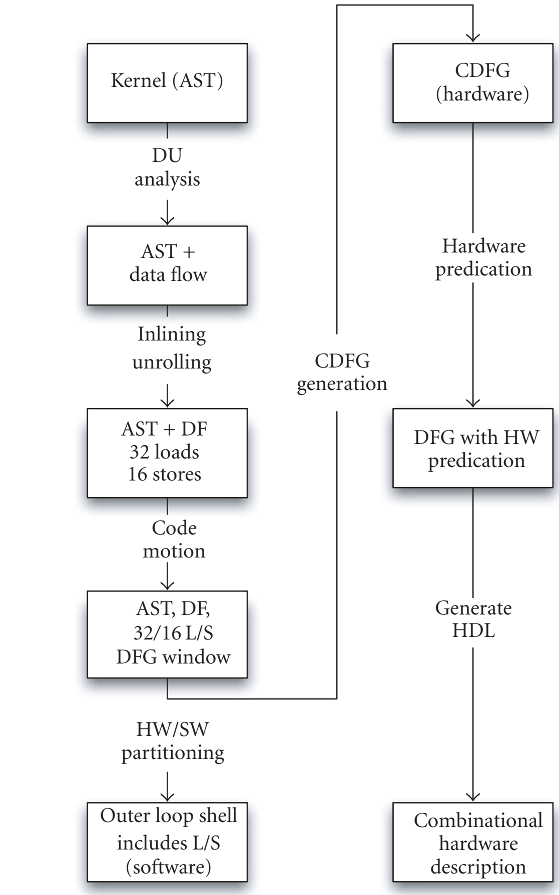


Figure 8: Description of the compilation and synthesis flow for portions of the code selected for custom hardware acceleration. Items on the left side are part of phase 1, which uses standard compiler transformations to prepare the code for synthesis. Items on the right side manipulate the code further using hardware predication to create a DFG for hardware implementation.

inputs. Sequential logic, on the other hand, requires knowledge of timing and prior inputs to determine the output values.

Our synthesis technique only relies on combinational logic synthesis and creates a *tractable* synthesis flow. The compiler generates data flow graphs (DFGs) that correspond to the computational kernel and, by directly translating these DFGs into a hardware description language like VHDL, these DFGs can be synthesized into entirely combinational logic for custom hardware execution using standard synthesis tools.

Figure 8 expands the behavioral synthesis block from Figure 7 to describe in more detail the compilation and synthesis techniques employed by our design flow to generate the hardware functions. The synthesis flow is comprised of two phases. Phase 1 utilizes standard compiler techniques operating on an abstract syntax tree (AST) to decouple loop control and memory accesses from the computation required by the kernel, which is shown on the left side of Figure 8. Phase 2 generates a CDFG representation of the computational code alone and uses *hardware predication* to convert this into a single DFG for combinational hardware synthesis.

```
fmult(int an, int srn) {
 short anmag, anexp, anmant;
 short wanexp, wanmag, wanmant;
short retval;
anmag = (an > 0) ? an: ((−an) & 0x1FFF);
anexp = quan(anmag, power2, 15) −6;
anmant = (anmag == 0) ? 32:
    (anexp >= 0) ? anmag >> anexp:
    anmag << −anexp;
wanexp = anexp + ((srn >> 6) & 0xF) −13;
wanmant = (anmant*(srn & 077)+0x30) >> 4;
retval = (wanexp >= 0) ?
        ((wanmant << wanexp) & 0x7FFF):
         (wanmant >> −wanexp);
return (((an^srn) < 0) ? −retval:
            retval);
}
```

Algorithm 3: Fmult function from G.721 benchmark.

5.2.1. Compiler transformations to restructure code

The kernel portion of the code is first compiled using the SUIF (Stanford University Intermediate Format) Compiler. This infrastructure provides an AST representation of the code and facilities for writing compiler transformations to operate on the AST. The code is then converted to SUIF2, which provides routines for definition-use analysis.

Definition-use (DU) analysis, shown as the first operation in Figure 8, annotates the SUIF2 AST with information about how the symbol (e.g., a variable from the original code) is used. Specifically, a *definition* refers to a symbol that is assigned a new value (i.e., a variable on the left-hand side of an assignment) and a *use* refers to an instance in which that symbol is used in an instruction (e.g., in an expression or on the right-hand side of an assignment). The *lifetime* of a symbol consists the time from the *definition* until the final *use* in the code.

The subsequent compiler pass, as shown in Figure 8, inlines functions within the kernel code segment to eliminate artificial basic block boundaries and unrolls loops to increase the amount of computation for implementation in hardware. The first function from Algorithm 2, predictor_zero(), calls the fmult() function shown in Algorithm 3. The fmult() function calls the quan() function which was also one of our top loops from Shark. Even though quan() is called (indirectly) by predictor_zero(), Shark provides execution for each loop independently. Thus, by inlining quan(), the subsequent code segment includes nearly 70% of the program's execution time. The computational kernel after function inlining is shown in Algorithm 4. Note that the local symbols from the inlined functions have been renamed by prepending the function name to avoid conflicting with local symbols in the caller function.

```
for (i = 0; i < 6; i++) {
    // begin fmult
    fmult_an = state_ptr−>b[i] >> 2;
    fmult_srn = state_ptr−>dq[i];
    fmult_anmag = (fmult_an > 0) ? fmult_an:
        ((−fmult_an) & 0x1FFF);
    // begin quan
    quan_table = power2;
    for (quan_i = 0; quan_i < 15; quan_i++)
      if (fmult_anmag < *quan_table++)
     break;
     fmult_anexp = quan_i;
     // end quan
     fmult_anmant = (fmult_anmag == 0) ? 32:
           (fmult_anexp >= 0) ?
           fmult_anmag >> fmult_anexp:
           fmult_anmag << −fmult_anexp;
     fmult_wanexp = fmult_anexp +
       ((fmult_srn >> 6) & 0xF) −13;
     fmult_wanmant = (fmult_anmant*
           (srn & 077)+0x30) >> 4;
     fmult_retval = (fmult_wanexp>= 0) ?
       ((fmult_wanmant<<fmult_wanexp) & 0x7FFF):
       (fmult_wanmant >> −fmult_wanexp);
     sezi += (((fmult_an^fmult_srn)< 0) ?
       −fmult_retval : fmult_retval);
     // end fmult
}
```

Algorithm 4: G.721 code after function inlining.

```
if (fmult_anmag < *quan_table)
    quan_i = 0;
else if (fmult_anmag < *(quan_table + 1))
    quan_i = 1;
else if (fmult_anmag < *(quan_table + 2))
    quan_i = 2;
...
else if (fmult_anmag < *(quan_table + 14)
    quan_i = 14;
```

Algorithm 5: Unrolled inner loop of inlined G.721 hardware kernel.

```
for (i = 0; i < 6; i++){
    quan_table_array_0 = *quan_table;
    quan_table_array_1 = *(quan_table + 1);
    ...
    quan_table_array_14 = *(quan_table + 14);
    state_pointer_b_array_i = state_ptr−>b[i];
    state_pointer_dq_array_i = state_ptr−>dq[i];
    // Begin Hardware Function
    fmult_an = state_pointer_b_array_i >> 2;
    fmult_srn = state_pointer_dq_array_i;
    if (fmult_anmag < quan_table_array_0)
       quan_i = 0;
    else if (fmult_anmag < quan_table_array_1)
       quan_i = 1;
    else if (fmult_anmag < quan_table_array_2)
       quan_i = 2;
    ...
    else if (fmult_anmag < quan_table_array_14)
       quan_i = 14;
    ...
    // End Hardware Function
}
```

Algorithm 6: G.721 benchmark after inlining, unrolling, and code motion compiler transformations. (Hardware functionality is in plain text with VLIW software highlighted with gray background.)

Once function inlining is completed, the inner loop is examined for implementation in hardware. By unrolling this loop, it is possible to increase the amount of code that can be executed in a single iteration of the hardware function. The number of loop iterations that can be unrolled is limited by the number of values that must be passed into the hardware function through the register file. In the example from Algorithm 4, each loop iteration requires a value loaded from memory, *quan_table, and a comparison with the symbol fmult_anmag. Because there are 15 iterations, complete unrolling results in a total of 16 reads from the register file. The resulting unrolled loop is shown in Algorithm 5. Once the inner loop is completely unrolled, the outer loop may be considered for unrolling. In the example, several values such as the array reads must be passed through the register file beyond the 16 required by the inner loop, preventing the outer loop from being unrolled. However, by considering a larger register file or special registers dedicated to hardware functions, this loop could be unrolled as well.

After unrolling and inlining is completed, there is a maximum of 32 values that can be read from the register file and 16 values that can be written to the register file. The next phase of the compilation flow uses code motion to move all memory loads to the beginning of the hardware function and move all memory stores to the end of the hardware function. This is done so as not to violate any data dependencies discovered during definition-use analysis. The loads from the unrolled code in Algorithm 5 are from the array quan_table that is defined prior to the hardware kernel code. Thus, loading the first 15 elements of quan_table array can be moved to the beginning of the hardware function code and stored in static symbols mapped to registers which the loops in the unrolled inner loop code. This is possible for all array accesses within the hardware kernel code for G.721. The hardware kernel code after code motion is shown in Algorithm 6.

As shown in Algorithm 6, the resulting code after DU analysis, function inlining, loop unrolling, and code motion is partitioned between hardware and software implementation. The partitioning decision is made statically such that all code required to maintain the loop (e.g., loop induction variable calculation, bounds checking and branching) and code required to do memory loads and stores is executed in software while the remaining code is implemented in hardware. This distinction is shown in Algorithm 6, where software code is highlighted with a gray background.

5.2.2. Synthesis of core computational code

Once hardware and software partitioning decisions are made as described in Section 5.2.1, the portion of the code for implementation in hardware is converted into a CDFG representation. This representation contains a series of basic blocks interconnected by control flow edges. Thus, each basic block boundary represents a conditional branch operation within the original code. Creation of a CDFG representation from a high level language is a well studied technique beyond the scope of this paper. However, details on creation of these graphs can be found in [6].

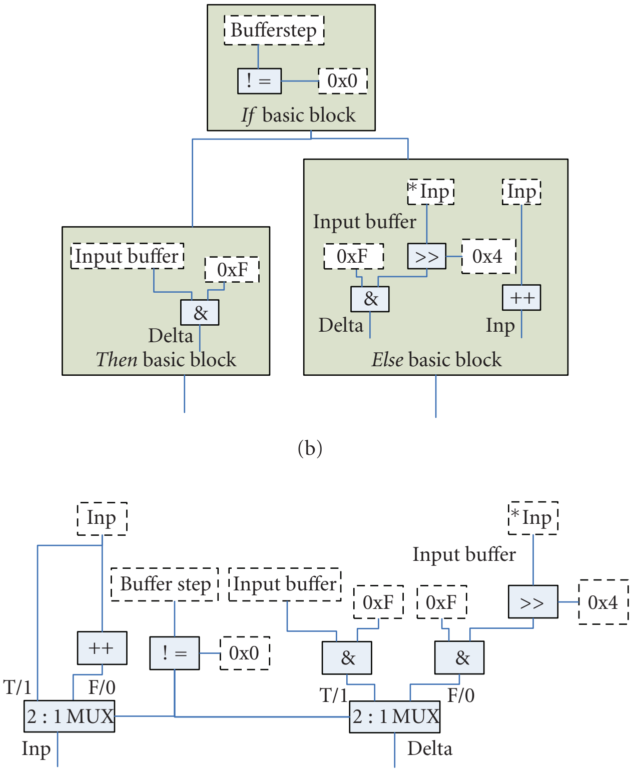
In order to implement the computation contained within the computational kernel, the control portions of the CDFG must be converted into data flow dependencies. This allows basic blocks, which were previously separated by control flow dependency edges to be merged into larger basic blocks which larger DFGs. If all the control flow dependencies can be successfully converted into data flow dependencies, the entire computational portion of the kernel can be represented as a single DFG. As a result, the DFG can be trivially transformed into a combinational hardware implementation, in our case using VHDL, and can be synthesized and mapped efficiently into logic within the target FPGA using existing synthesis tools.

Our technique for converting these control flow dependencies into data flow dependencies is called *hardware predication*. This technique is similar to ADDs developed as an alternate behavioral representation for synthesis flows, see Section 3. Consider a traditional *if-then-else* conditional construct written in C code. In hardware, an *if-then-else* conditional statement can be implemented using a multiplexer acting as a binary switch to predicated output datapaths. To execute the same code in software, an *if-then-else* statement is implemented as a stream of six instructions composed of comparisons and branch statements. Figure 9 shows several different representations of a segment of the kernel code from the ADPCM encoder benchmark. Figure 9(a) lists the assembly code, Figure 9(b) shows the corresponding CDFG representation of the code segment, and Figure 9(c) presents a data flow diagram for a 2 : 1 hardware predication (e.g., multiplexer) equivalent of the CDFG from Figure 9(b).

In the example from Figure 9, the *then* part of the code from Figure 9(a) is converted into the *then* basic block Figure 9(b). Likewise the statements from the *else* portion in Figure 9(a) are converted in into the *else* basic block in Figure 9(b). The CDFG in Figure 9(b) shows that the control flow from the *if-then-else* construction creates basic block boundaries with control flow edges. The hardware predication technique converts these control flow dependencies into data flow dependencies allowing the CDFG in Figure 9(b) to be transformed into the DFG in Figure 9(c). Each symbol with a definition in either or both of the basic blocks following the conditional statement (i.e., the *then* and *else* blocks from Figure 9(b)) must be predicated by inserting a multiplexer. For example, in Figure 9, the symbol delta is defined in both blocks and these definitions become inputs to a rightmost selection multiplexer in Figure 9(c). The symbol inp is updated in the *else* basic block only in Figure 9(b). This requires the leftmost multiplexer in Figure 9(c), where the original value from prior to the condition and the updated value from the *else* block become inputs. All of the multiplexers instantiated due to the conversion of these control flow edges into data flow edges are based on the conditional operation from the *if* basic block in Figure 9(b).

```
If (bufferstep){
delta = inputbuffer & 0xf;
} else {
inputbuffer =*inp++
delta = (inputbuffer >> 4) & 0xf;
}
```

(a)



(b)

(c)

FIGURE 9: Software code, CDFG, and DFG with predicated hardware example for control flow in ADPCM encoder.

By implementing the logic in this manner, the six clock cycles required for execution in the processor can be reduced to two levels of combinational logic in hardware. Considering the example of Figure 9, the assembly code requires as many as nine (9) cycles if the *else* path is selected, but the hardware version can be implemented as two levels of combinational logic (constant shifts are implemented as wires).

In many cases, this type of hardware predication works in the general case and creates efficient combinational logic for moderate performance results. However, in some special cases, control flow can be further optimized for combinational hardware implementation. In C, switch statements, sometimes called multiway branches, can be handled specially. While this construct can be interpreted in sequence to execute the C code, directly realizing this construct with multiplexing hardware containing as many inputs as cases in the original code allows entirely combinational, parallel execution. A second special case exists for the G.721 example described in Section 5.2.1. Consider the unrolled innermost loop shown in Algorithm 6. This code follows the construction if (cond), else if (cond2), . . . , else if (condN). This is similar to the behavior of a priority encoder in combinational hardware where each condition has a priority, such as high bit significance overriding lower bit significance. For example, in a one-hot priority encoder, if the most significant bit (MSB) is "1", then all other bits are ignored and treated as zeros. If the MSB is "0" and the next MSB is "1", then all other bits are assumed "0." This continues down into the least significant bit. When this type of conditional is written in a similar style in synthesizable HDL, synthesis tools will implement a priority encoder, just like a case statement in HDL implements a multiplexer. Thus, for the cases where this type of code is present for either the multiplexer or the priority encoder, this structure is retained.

5.3. *Interfacing hardware and software*

A hardware function can be called with no additional overhead requirements versus that of executing the code directly in software. The impact of even a small hardware/software overhead can dramatically reduce the speedup that the kernel achieves. In essence, some of the speed benefit gained from acceleration is lost due to the interface overhead.

Consider (1), β is the *hardware speedup* defined as the ratio of software to hardware execution time. This equation only considers hardware acceleration and does not equate directly to *kernel speedup*. In (2), α is the actual kernel speedup as this considers the portion of the kernel that cannot be translated to hardware. This is labeled as overhead (OH). This definition is actually a misnomer as it implies that there is an *added* overhead for running our kernel hardware. In fact, this "overhead" is actually the same loads and stores that would be run in the software only solution. No additional computation is added,

$$\beta = \frac{t_{\mathrm{sw}}}{t_{\mathrm{hw}}}, \tag{1}$$

$$\alpha = \frac{t_{\mathrm{sw}}}{t_{\mathrm{OH}} + t_{\mathrm{hw}}} = \frac{\beta}{t_{\mathrm{OH}}/t_{\mathrm{hw}} + 1}. \tag{2}$$

Figure 10 shows the effect of adding 0 to 128 cycles of hardware/software overhead on a set of hardware accelerated kernels. We explain how these speedups are achieved later on in this paper and focus here on the impact of data movement overhead. A zero overhead is the pure speedup of the hardware versus the software. Note that even 2 software cycles of overhead, perhaps caused by a single I/O write and one I/O read, causes the effective kernel speedup to be cut down by a half. For a bus-based system, tens of processor cycles of latency dramatically diminish the benefit of hardware acceleration. Thus, by enabling direct data sharing through the register file, our architecture does not incur any penalty.

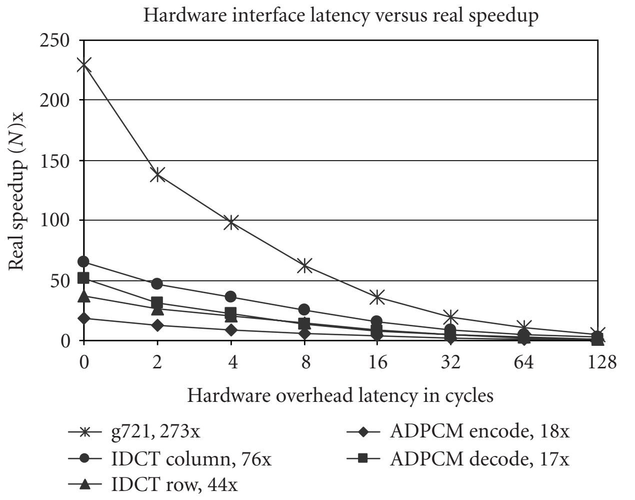


Figure 10: Real speedup of hardware benchmark functions compared to software execution given varying interface latencies.

Table 2: Execution profile of benchmarks.

	Execution time of		
Benchmark	*Kernel 1*	*Kernel 2*	*Total*
ADPCM decode	99.9%	N/A	99.9%
ADPCM encode	99.9%	N/A	99.9%
G.721 decode	70.5%	N/A	70.5%
GSM decode	71.0%	N/A	71.0%
MPEG 2 decode	21.5%	21.4%	42.9%

6. BENCHMARKS

To evaluate the effectiveness of our approach for signal processing applications, we selected a set of core signal processing benchmarks. We examine algorithms of interest in signal processing from three categories: voice compression, image and video coding, and wireless communication. The following sections describe selected benchmarks in these domains and specifically examine benchmark codes selected from each domain. Except for the so-called sphere decoder, the software codes examined in the following sections were taken from the MediaBench benchmark suite [57].

Table 2 shows the execution time contribution of the computational kernels from signal processing oriented benchmarks from MediaBench. For example, ADPCM encode and decode kernels contribute nearly the entirety of the application execution time. Both the G.721 and GSM benchmarks top kernel requires over 70% of the execution time

Table 3: Instruction level parallelism (ILP) extracted using the Trimaran compiler.

	Instruction level parallelism			
Benchmark	*Kernel 1*	*Kernel 2*	*Nonkernel*	*Avg*
ADPCM decode				
4-way VLIW	1.13	N/A	1.23	1.18
Unlimited VLIW	1.13	N/A	1.23	1.18
ADPCM encode				
4-way VLIW	1.28	N/A	1.38	1.33
Unlimited VLIW	1.28	N/A	1.38	1.33
G.721 decode				
4-way VLIW	1.25	N/A	1.32	1.28
Unlimited VLIW	1.41	N/A	1.33	1.37
GSM decode				
4-way VLIW	1.39	N/A	1.25	1.32
Unlimited VLIW	1.39	N/A	1.25	1.32
MPEG 2 decode				
4-way VLIW	1.68	1.40	1.41	1.54
Unlimited VLIW	1.84	1.50	1.46	1.67

and MPEG 2 decoder requires two separate loop kernels to achieve between less than 50% of the execution times.

The ILP of the benchmarks is shown in Table 3. The ILP numbers are broken into three groups, the first being the ILP for the computational kernel of highest complexity (kernel 1), the second for the next highest kernel (kernel 2), which is only necessary for the MPEG 2 benchmark, the ILP of the nonkernel software code, and finally, a nonweighted average ILP for the entire application. All numbers were reported for both a standard 4-way VLIW processor as implemented in our system and compared with numbers for a theoretical unlimited-way VLIW processor.

This limited ILP shows that VLIW processing alone can only provide a nominal performance improvement. The range of speedups possible will be of 20–60% overall, which is far below our target for these applications. To discover how speedups can be achieved through hardware functions in our system, we begin by examining our algorithms, specifically the computational kernel codes below.

6.1. Voice compression algorithms

We chose three representative voice compression algorithms as benchmarks. These were drawn from various application areas in voice compression and reflect quite different coding algorithms. In each case, the C-language implementation benchmark came from the MediaBench suite. We have purposefully chosen well-established implementations to demonstrate the practical performance gains immediately available to the system designer through our approach.

The first system we examined was the International Consultative Committee on Telegraphy and Telephony (CCITT) G.721 standard, which employs adaptive differential pulse code modulation (ADPCM) to compress toll quality audio signals down to 32 *kpbs* [57]. The G.721 audio compression standard is employed in most European cordless telephones.

We next consider CCITT-ADPCM, a different ADPCM implementation that is recommended by the IMA Digital Audio Technical Working Group. The algorithm takes 16 bit PCM samples and compresses them to four bit ADPCM samples, generating a compression ratio of 4 : 1.

The last speech compression algorithm we consider is the GSM 06.10 standard specified for use with the global system for mobile telecommunication (GSM) wireless standard. In the GSM 06.10 standard, residual pulse excitation/long term prediction (RPELTP) is used to encode the speech signal at a compression ratio of 8 : 1. The linear prediction engine runs Schur recursions, which was argued by the package designer to yield some performance advantages over the usual Levinson-Durbin algorithm when parallelized [58].

One of the significant bottlenecks of increasing algorithmic execution is control flow requirements (e.g., determining the next operation to execute based on the result of previous operations). Algorithms high in control flow map very well to sequential processors as these processors are highly optimized to execute these sequential codes by achieving high throughputs and clock speeds through techniques like pipelined execution.

When implementing heavily control-oriented codes in hardware, sequential structures such as finite state machines (FSMs) are often used for this purpose. Unfortunately, these FSMs do not allow significantly more parallelism than running this code in a processor. To achieve a speedup using a VLIW processor it is necessary to attempt to remove the control flow dependencies to allow parallel execution. In sequential processors, predication is used to convert many types of control flow to data flow dependencies.

Consider the ADPCM encoder shown in Algorithm 7. The for loop in the example consumes nearly 100% of the execution time (see Table 2). Excluding the control flow associated with the for loop, this code segment contains nine (9) conditional executions. These statements are candidates for predication.

To allow predicated execution in a processor, one or more *predication registers* are used. Conditional branch instructions are traditionally used to execute *if* statements. To use predication, these branch instructions are replaced by conditional operations followed by predicated instructions. For example, in Algorithm 7, line 7, the subtraction operation is only executed if diff >= step. Thus, the conditional is calculated and the result is stored in a predication register. The subtraction instruction can be issued and the result will only be saved if the conditional is true. The same predication register can also be used for the addition operation in line 8. This type of predication allows increased ILP and reduces stalls in pipelined execution.

One of the restrictions we place on our hardware functions is that they consist entirely of combinational logic (e.g., they do not require sequential execution). As a result, we use a technique related to predication called parallel execution.

```
1. for ( ; len > 0; len–) {
2. val = *inp++;
3. delta = 0;
4. vpdiff = (step >> 3);
5. if (diff >= step) {
6.    delta = 4;
7.    diff − = step;
8.    vpdiff += step;
9. }
10. step >>= 1;
11. if (diff >= step) {
12.    delta | = 2;
13.    diff − = step;
14.    vpdiff += step;
15. }
16. step >>= 1;
17. if (diff >= step) {
18.    delta | = 1;
19.    vpdiff += step;
20. }
21. if (sign) valpred − = vpdiff;
22. else valpred += vpdiff;
23. if (valpred > 32767)
24.    valpred = 32767;
25. else if (valpred < −32768)
26.    valpred = −32768;
27. delta | = sign;
28. index += indexTable[delta];
29. if (index < 0) index = 0;
30. if (index > 88) index = 88;
31. step = stepsizeTable[index];
32. if (bufferstep) {
33.    outputbuffer = (delta << 4) & 0xf0;
34. } else {
35.    *outp++ = (delta & 0x0f) | outputbuffer;
36. }
37. bufferstep = !bufferstep;
38. }
```

Algorithm 7: ADPCM encoder kernel C code. (Hardware functionality is in plain text with VLIW software highlighted with gray.)

For an if statement, both the *then* and *else* parts of the statement are executed and propagated down the DFG based on the result of the conditional. For example, the ADPCM encoder from Algorithm 7 was translated into the DFG shown in Figure 11. The blocks labelled *MUX* implement the combinational parallel execution. The conditional operation is used as the selector and the two inputs contain the result of the "predicated" operation as well as the nonmodified result.

Two other standard automation techniques were used to convert the code segment into the DFG. First, the load from memory *inp from line 2 and the predicated store *outp from line 34 of Algorithm 7 are moved to the beginning and end of the DFG, respectively, using code motion. This allows the loads and stores to be executed in software. All code executed in software is highlighted in Algorithm 7. Secondly the static arrays indexTable and stepsizeTable are converted into lookup-tables (LUTs) for implementation in ROM structures.

The computational kernel source code for the G.721 benchmark was used in the prior section to describe the various design automation phases. The resulting DFG for the G.721 based on the transformations is displayed in Figure 12. It should be noted that the completely unrolled loop has been transformed into a priority encoder in the hardware implementation. It can also be seen that aside from the encoder, there is only a moderate amount of computations.

6.2. The discrete cosine transform

We next consider a hardware implementation of the inverse discrete-time cosine transform (IDCT). The IDCT arises in several signal processing applications, most notably in image/video coding (see, e.g., the MPEG standard) and in more general time-frequency analysis. The IDCT is chosen because there has been a large amount of work on efficient algorithm design for such transforms. Our results argue that further gains are possible with relatively little additional design overhead by employing a mixed architecture.

The IDCT code from the MediaBench suite was extracted from the MPEG 2 benchmark and specifically from the MPEG 2 Decoder used in a variety of applications, most notably for DVD movies. The IDCT actually appears in the JPEG benchmark; however, the implementation from MPEG 2 was selected because it has the longer runtime. The implementation in MPEG 2 decomposes the IDCT into a row-wise and columnwise IDCT.

The IDCT algorithm does a two-dimesional IDCT through decomposition. It first executes a one-dimensional DCT in one dimension (row) followed a one-dimensional DCT in the other (column). The IDCT columnwise decomposition kernel with the software portions highlighted is shown in Algorithm 8.

Like the IDCT row-wise decomposition, the DFG in Figure 13 again contains a significant number of arithmetic functional units and shows a significant amount of parallelism.

6.3. The sphere decoder

The last application example we consider is the so-called sphere decoder, which arises in MIMO communication systems. The basic MIMO problem is the following: an *M-tuple* of information symbols **s**, drawn from the integer lattice $\mathbb{Z}^M$, is transmitted from M transmit antennas to N receive antennas. We assume that the channel is "flat," meaning that the fading parameter connecting transmit antenna m to receive antenna n, $h_{m,n,}$ can be modeled as a scalar, constant over the transmission. The model at the output of a bank of receiver matched filters (one per receive antenna) is simply

$$\mathbf{y} = \mathbf{H}\mathbf{s} + \mathbf{n}, \tag{3}$$

where **n** modeled as zero mean additive white Gaussian noise (AWGN) arising from receiver electronics noise and possibly from channel interference. We assume that the receiver can track the channel coefficients. Note that the use

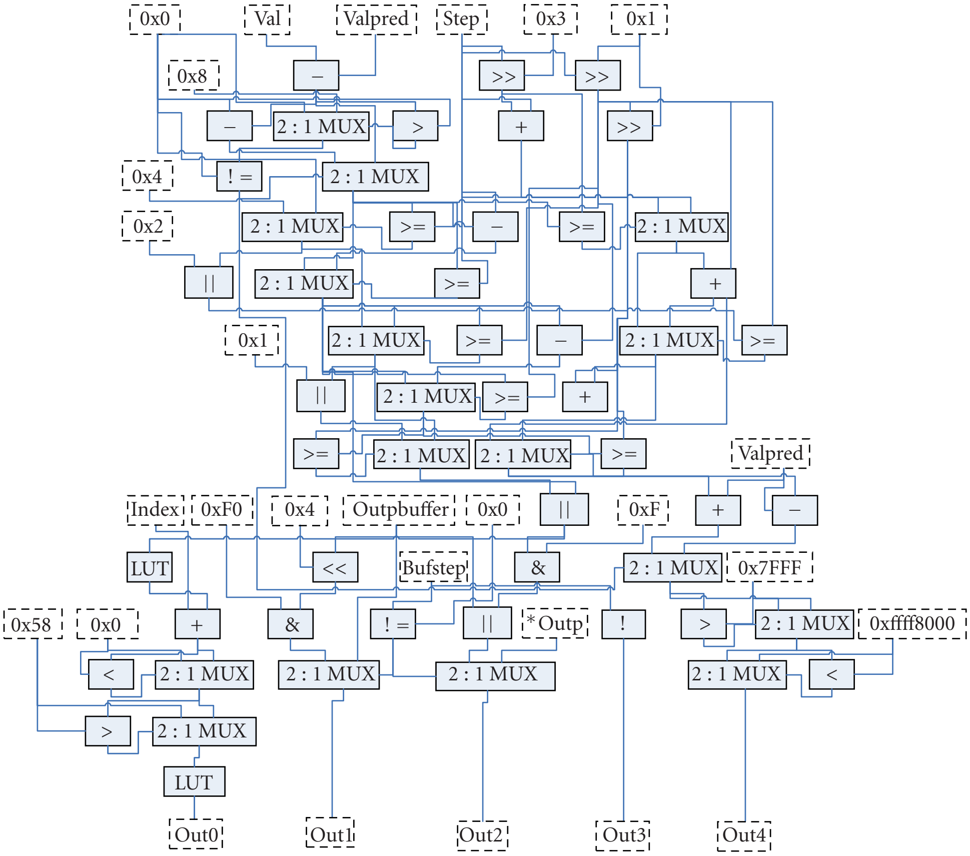


Figure 11: Data flow graph for the ADPCM encoder.

of a real-valued channel model loses no generality since any complex model employing quadrature-amplitude modulation (QAM) at the transmitter can be reduced to a real model of twice the dimensions (see, e.g., [59]).

At the receiver we seek to find the input **b** which minimizes the detection norm

$$\hat{\mathbf{s}} = \arg\min_{s} \|\mathbf{H}\mathbf{s} - \mathbf{y}\|^2, \tag{4}$$

which is generally an exponentially hard problem (i.e., we must consider Q^M possible signals when an alphabet of size Q is employed). The sphere decoder employs the Fincke-Pohst tree-based search algorithm over Z^M to reduce this to a detection rule which is roughly *cubic* in M for sufficiently large signal-to-noise ratios [60, 61]. It is expected to form the core of practical receivers for future MIMO systems.

Unlike the previous algorithms, the spherical decoder was written in vectorized Matlab rather than C. Matlab is a popular language for signal processing algorithm development because of its native matrix treatment and its built-in mathematical functions. Vectorized Matlab has another nice feature; it is easy to use for extracting parallelism, a main feature for achieving performance improvements by direct hardware implementation.

The code from Algorithm 9 represents the computational core of the spherical decoder, by executing a pointwise vector multiplication, a vector summation, and finally a distance calculation. The DFG for this function is displayed in Figure 14 that includes two square functions and a square-root.

7. RESULTS

To implement our architecture including the hardware functions, several industry computer-aided design tools were used to accomplish the functional testing and gate-level synthesis tasks of the design flow. We used Mentor Graphics' FPGA Advantage 6.1 to assist in the generation of VHDL code used to describe the core processor architecture. The processor was built up to support all of the operations described in the NIOS II instruction set.

We used Synplicity's Synplify Pro 7.6.1 as the RTL synthesis tool to generate the gate-level netlist targeted to the Altera Stratix II ES2S180F1508C4 from our VHDL description. The netlist was then passed to Altera's Quartus II 4.1 for device-specific, placement and routing, and bitstream generation. At this level, post placement and routing results were extracted for additional manual optimization, timing accurate simulation, and verification. It is at this point that the timing information about the hardware functions can be inserted into the software. Both Altera's Quartus and Synplicity's Amplify for

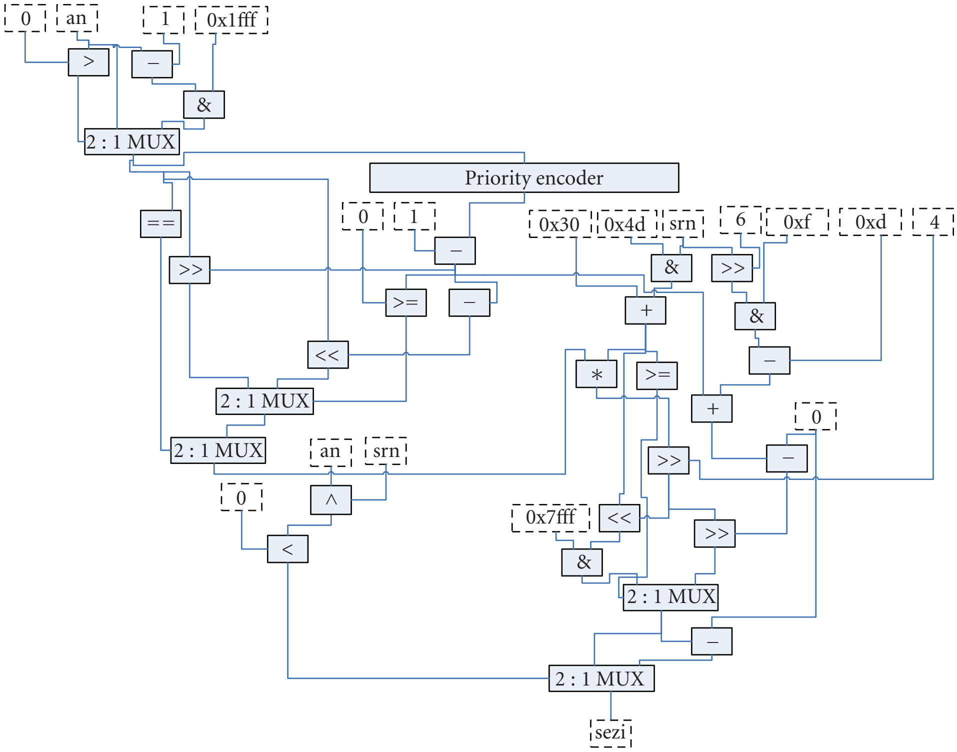


FIGURE 12: Resulting DFG from transformations described in G.721 example.

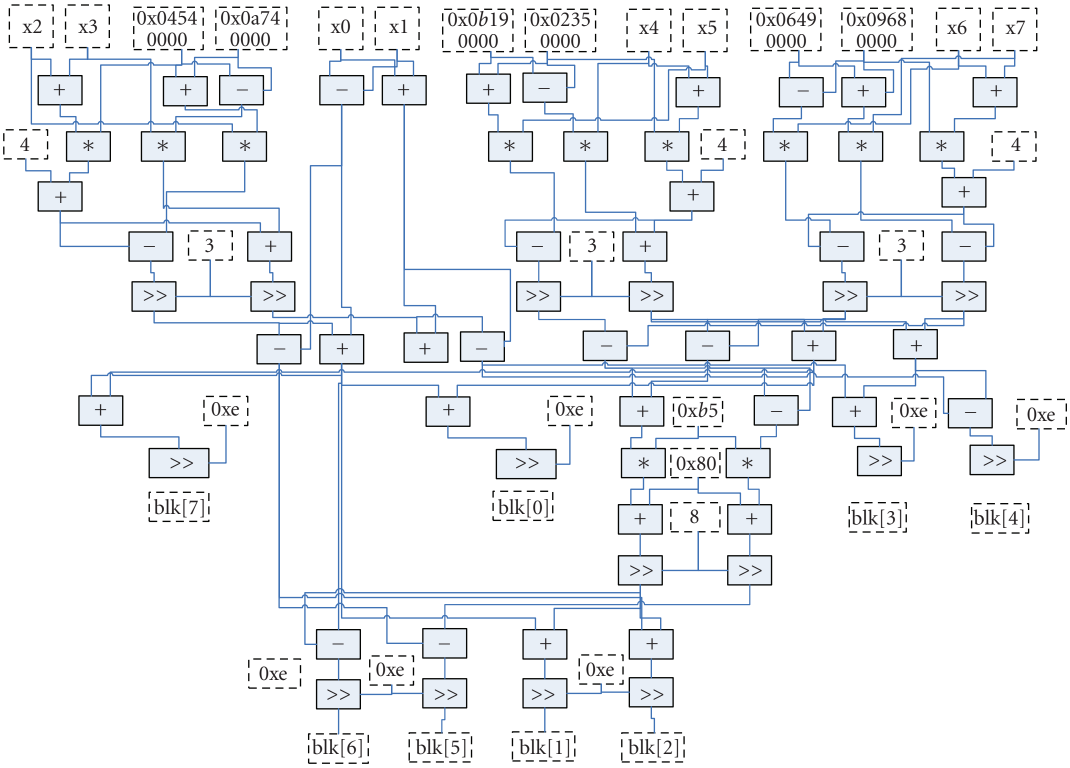


FIGURE 13: IDCT column-wise decomposition data flow graph.

```
if (!((x1 = (blk[8 * 4] << 8))
   | (x2 = blk[8 * 6]) | (x3 = blk[8 * 2])
   | (x4 = blk[8 * 1]) | (x5 = blk[8 * 7])
   | (x6 = blk[8 * 5]) | (x7 = blk[8 * 3]))){
    blk[8 * 0] = blk[8 * 1] = blk[8 * 2] = blk[8 * 3] =
    blk[8 * 4] = blk[8 * 5] = blk[8 * 6] = blk[8 * 7] =
    iclp[(blk[8 * 0] + 32) >> 6];
    return;
  }
x0 = (blk[8 * 0] << 8) + 8192;
/* first stage */
x8 = W7 * (x4 + x5) + 4;
x4 = (x8 + (W1 - W7) * x4) >> 3;
x5 = (x8 - (W1 + W7) * x5) >> 3;
x8 = W3 * (x6 + x7) + 4;
x6 = (x8 - (W3 - W5) * x6) >> 3;
x7 = (x8 - (W3 + W5) * x7) >> 3;
/* second stage */
x8 = x0 + x1;
x0 - = x1;
x1 = W6 * (x3 + x2) + 4;
x2 = (x1 - (W2 + W6) * x2) >> 3;
x3 = (x1 + (W2 - W6) * x3) >> 3;
x1 = x4 + x6;
x4- = x6;
x6 = x5 + x7;
x5- = x7;
/* third stage */
x7 = x8 + x3;
x8- = x3;
x3 = x0 + x2;
x0- = x2;
x2 = (181 * (x4 + x5) + 128) >> 8;
x4 = (181 * (x4 - x5) + 128) >> 8;
/* fourth stage */
blk[8 * 0] = iclp[ (x7 + x1) >> 14 ] ;
blk[8 * 1] = iclp[ (x3 + x2) >> 14 ] ;
blk[8 * 2] = iclp[ (x0 + x4) >> 14 ] ;
blk[8 * 3] = iclp[ (x8 + x6) >> 14 ] ;
blk[8 * 4] = iclp[ (x8 - x6) >> 14 ] ;
blk[8 * 5] = iclp[ (x0 - x4) >> 14 ] ;
blk[8 * 6] = iclp[ (x3 - x2) >> 14 ] ;
blk[8 * 7] = iclp[ (x7 - x1) >> 14 ] ;
```

Algorithm 8: IDCT column-wise decomposition kernel. (Hardware functionality is in plain text with VLIW software highlighted with gray.)

FPGA allow manual routing modifications for optimization of design density and critical paths.

For functional simulation and testing of the design, we passed the machine code output from the compiler design flow into the instruction ROM used in modeling our design. ModelSim SE 5.7 was used to generate the waveforms to confirm the functional correctness of our VLIW processor with hardware function acceleration.

```
k = k - 1;
ym(k,k + 1) = y(k) - sum(R(k,k + 1 : M) * s(k + 1 : M));
rp(k) = sqrt(rp(k + 1)^2 - (ym(k + 1,k + 2)-
        R(k + 1,k + 1) * s(k + 1))^2);
```

Algorithm 9: Matlab kernel code for the spherical decoder.

Through a series of optimizations to the critical path, we were able to achieve a maximum clock speed of 166 MHz for our VLIW and clock frequencies ranging from 22 to 71 MHz for our hardware functions equating to combinational delays from 14 to 45 ns. We then compared benchmark execution times of our VLIW both with and without hardware acceleration against the pNIOS II embedded soft-core processor.

To exercise our processor, we selected core signal processing benchmarks from the MediaBench suite that include ADPCM encode and decode, GSM decode, G.721 decode, and MPEG 2 decode. As described in Section 6, from each of the benchmarks a single hardware kernel was extracted with the exception of MPEG 2 decode for which two kernels were extracted. In the case of GSM and G.721, the hardware kernel was shared by the encoder and decoder portions of the algorithm.

The performance improvement of implementing the computational portions of the benchmark on a 4-way VLIW, an unlimited-way VLIW, and directly in the hardware compared to a software implementation running on pNIOS II is displayed in Figure 15. The VLIW performance improvements were fairly nominal ranging from 2% to 48% improvement over pNIOS II, a single ALU RISC processor. The performance improvement of the entire kernel execution is compared for a variety of different architectures in Figure 16. The difference between Figures 15 and 16 is that the loads and stores required to maintain the data in the register file are not considered in the former and run in software in the latter. When the software-based loads and stores are considered, the VLIW capability of our processor has a more significant impact. Overall kernel speedups range from about 5X to over 40X.

The width of the available VLIW has a significant impact on the overall performance. In general, the 4-way VLIW is adequate, although particularly when considering the IDCT-based benchmark, the unlimited VLIW shows that not all of the ILP available is being exploited by the 4-way VLIW.

The performance of the entire benchmark is considered in Figure 17. We compare execution times for both hardware and VLIW acceleration and compare them to the pNIOS II processor execution when the overheads associated with hardware function calls are included. While VLIW processing alone again provides nominal speedup (less than 2X), by including hardware acceleration, these speedups range from about 3X to over 12X and can reach nearly 14X when combined with a 4-way VLIW processor.

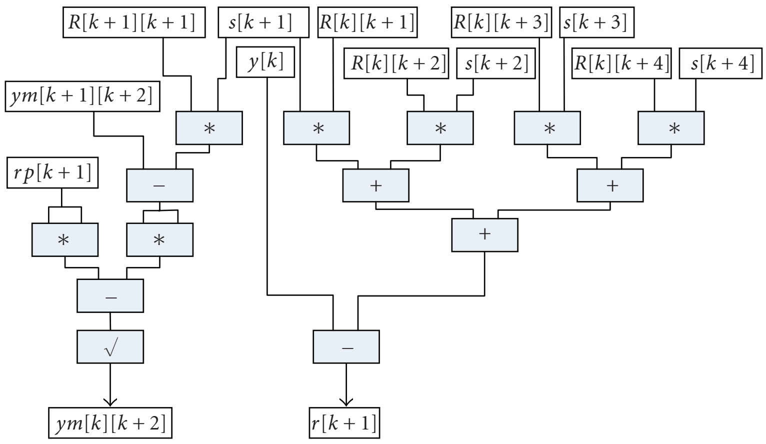
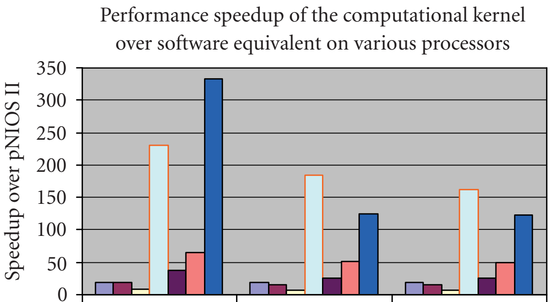
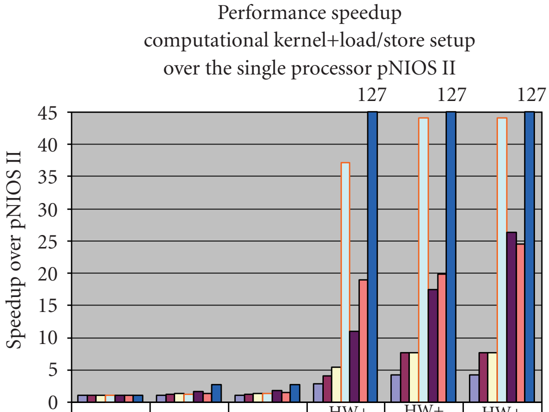


FIGURE 14: Data flow graph for the spherical decoder from Matlab.



	pNIOS II	VLIW 4	VLIW Unl
■ ADPCM decoder	18.33	18	18
■ ADPCM encoder	18.25	16	16
□ GSM decoder	9.33	7	7
□ G721 decoder	230	184	161.67
■ IDCT row	37.40	25.20	25.20
■ IDCT col	64.80	51.20	50
■ Spherical decoder	332.50	124.13	123.25

FIGURE 15: Performance improvement from hardware acceleration of computational portion of the hardware kernel.



	pNIOS II	VLIW 4	VLIW Unl	HW+ pNIOS II	HW+ VLIW 4	HW+ VLIW Unl
■ ADPCM decoder	1	1.13	1.13	2.93	4.16	4.16
■ ADPCM encoder	1	1.28	1.28	4	7.67	7.67
□ GSM decoder	1	1.39	1.39	5.41	7.67	7.67
□ G721 decoder	1	1.25	1.41	37.16	44.13	44.13
■ IDCT row	1	1.68	1.84	10.96	17.53	26.30
■ IDCT col	1	1.40	1.50	18.95	19.86	24.53
■ Spherical decoder	1	2.66	2.68	127.29	127.29	127.29

FIGURE 16: Kernel speeds up several architectures when considering required loads and stores to maintain the register file.

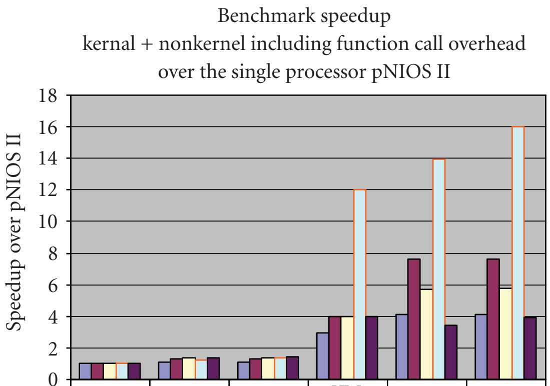
8. CONCLUSIONS AND FUTURE WORK

In this paper, we describe a VLIW processor with the capability of implementing computational kernels in custom combinational hardware functions with no additional overhead. We provide a design methodology to map algorithms written in C onto this processor and utilize profiling and tractable behavioral synthesis to achieve application speedups.

We tested our processor with a set of signal processing benchmarks from the MediaBench suite and achieved a hardware acceleration of 9.3 to 332x with an average of 63X better than a single processor, depending on the kernel. For the entire application, the speedup reached nearly 30X and was on average 12X better than a single processor implementation.

VLIW processing alone was shown to achieve very poor speedups, reaching less than 2X maximum improvement even for an unlimited VLIW. This is due to a relatively small improvement in ILP and, thus, provides relatively small performance improvement. However, when coupled with hardware functions, the VLIW has a significant impact providing, in some cases, up to an additional 3.6X. It provided an average of 2.3X over a single processor and hardware alone. This range falls within the additional predicted 2X to 5X processing capability provided by the 4-way VLIW processor.

The reason for this improvement is due to several factors. The first is a simplified problem. Because the software code has been made more regular (just loading and storing non-data-dependent values), the available ILP for VLIW processing is potentially much higher than in typical code. Secondly, we see a *new* "90/10" rule. The hardware execution



	pNIOS II	VLIW 4	VLIW Unl	HW+ pNIOS II	HW+ VLIW 4	HW+ VLIW Unl
■ ADPCM decoder	1	1.13	1.13	2.92	4.15	4.15
■ ADPCM encoder	1	1.28	1.28	4	7.66	7.66
□ GSM decoder	1	1.35	1.36	4.02	5.71	5.77
□ G721 decoder	1	1.27	1.39	12.01	13.95	16.01
■ MPEG2 decode	1	1.39	1.48	3.97	3.42	3.95

Figure 17: Overall performance speedup of the entire application for several architectures including overheads associated with function calls.

accelerates a high percentage of the kernel code by 9X or more leaving the remaining software portion of the code to dominate the execution time. Improving the remaining software code execution time through VLIW processing impacts this remaining (and now dominant) execution time, thus providing magnified improvements for relatively low ILP (such as the predicted 2–5X).

While the initial results for the VLIW processor with hardware functions are encouraging, there are several opportunities for improvement. A significant limiting factor of the hardware acceleration is the loads and stores that are currently executed in software. While these operations would need to be done in software for a single processor execution and it is possible through transformations that these operations have become more regular and, thus, exhibit higher than normal ILP, they still are a substantial bottleneck.

To improve the performance of these operations it should be possible to overlap load and store operations with the execution of the hardware block. One way to do this is to create "mirror" register files. While the hardware function executes on one group of registers, the VLIW prepares the mirror register file for the next iteration. Another possibility to allow the hardware direct access to the memories as well as the register file.

ACKNOWLEDGMENT

This work was supported in part by the Pittsburgh Digital Greenhouse.

REFERENCES

[1] Altera Corporation, "Stratix II Device Handbook, Volume 1," available on-line: http://www.altera.com.

[2] Xilinx Incorporated, "Virtex-4 Product Backgrounder," available on-line: http://www.xilinx.com.

[3] Lattice Semiconductor Corporation, "LatticeECP and EC Familiy Data Sheet," available on-line: http://www.latticesemi.com.

[4] Apple Computer Inc., "Optimizing with SHARK, Big Payoff, Small Effort".

[5] D. C. Suresh, W. A. Najjar, F. Vahid, J. R. Villarreal, and G. Stitt, "Profiling tools for hardware/software partitioning of embedded applications," in *Proceedings of ACM SiGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '03)*, pp. 189–198, San Diego, Calif, USA, June 2003.

[6] G. De Micheli, D. Ku, F. Mailhot, and T. Truong, "The Olympus synthesis system," *IEEE Design and Test of Computers*, vol. 7, no. 5, pp. 37–53, 1990.

[7] L. Lavagno and E. Sentovich, "ECL: a specification environment for system-level design," in *Proceedings of 36th Design Automation Conference (DAC '99)*, pp. 511–516, New Orleans, La, USA, June 1999.

[8] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK: a high-level synthesis framework for applying parallelizing compiler transformations," in *Proceedings of 16th IEEE International Conference on VLSI Design (VLSI Design '03)*, pp. 461–466, New Delhi, India, January 2003.

[9] S. Gupta, N. Savoiu, N. Dutt, R. Gupta, and A. Nicolau, "Using global code motions to improve the quality of results for high-level synthesis," *IEEE Transactions On Computer-Aided Design Of Integrated Circuits and Systems*, vol. 23, no. 2, pp. 302–312, 2004.

[10] A. K. Jones, D. Bagchi, S. Pal, P. Banerjee, and A. Choudhary, "Pact HDL: compiler targeting ASIC's and FPGA's with power and performance optimizations," in *Power Aware Computing*, R. Graybill and R. Melhem, Eds., chapter 9, pp. 169–190, Kluwer Academic, Boston, Mass, USA, 2002.

[11] X. Tang, T. Jiang, A. K. Jones, and P. Banerjee, "Behavioral synthesis of data-dominated circuits for minimal energy implementation," in *Proceedings of 18th IEEE International Conference on VLSI Design (VLSI Design '05)*, pp. 267–273, Kolkata, India, January 2005.

[12] E. Jung, "Behavioral synthesis using systemC compiler," in *Proceedings of 13th Annual Synopsys Users Group Meeting (SNUG '03)*, San Jose, Calif, USA, March 2003.

[13] D. Black and S. Smith, "Pushing the limites with behavioral compiler," in *Proceedings of 9th Annual Synopsys Users Group Meeting (SNUG '99)*, San Jose, Calif, USA, March 1999.

[14] K. Bartleson, "A New Standard for System-Level Design," Synopsys White Paper, 1999.

[15] R. Goering, "Behavioral Synthesis Crossroads," EE Times Article, 2004.

[16] D. J. Pursley and B. L. Cline, "A practical approach to hardware and software SoC tradeoffs using high-level synthesis for architectural exploration," in *Proceedings of of the GSPx Conference*, Dallas, Tex, USA, March–April 2003.

[17] S. Chappell and C. Sullivan, "Handel-C for Co-Processing and Co-Design of Field Programmable System on Chip," Celoxica White Paper, 2002.

[18] P. Banerjee, M. Haldar, A. Nayak, et al., "Overview of a compiler for synthesizing MATLAB programs onto FPGAs," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 3, pp. 312–324, 2004.

[19] P. Banerjee, N. Shenoy, A. Choudhary, et al., "A MATLAB compiler for distributed, heterogeneous, reconfigurable computing systems," in *Proceedings of 8th Annual IEEE International*

Symposium on FPGAs for Custom Computing Machines (FCCM '00), pp. 39–48, Napa Valley, Calif, USA, April 2000.

[20] S. McCloud, "Catapult C Synthesis-Based Design Flow: Speeding Implementation and Increasing Flexibility," Mentor Graphics White Paper, 2004.

[21] V. Chaiyakul and D. D. Gajski, "Assignment decision diagram for high-level synthesis," Tech. Rep. #92-103, University of California, Irvine, Calif, USA, December 1992.

[22] V. Chaiyakul, D. D. Gajski, and L. Ramachandran, "High-level transformations for minimizing syntactic variances," in *Proceedings of 30th Design Automation Conference (DAC '93)*, pp. 413–418, Dallas, Tex, USA, June 1993.

[23] I. Ghosh and M. Fujita, "Automatic test pattern generation for functional RTL circuits using assignment decision diagrams," in *Proceedings of 37th Design Automation Conference (DAC '00)*, pp. 43–48, Los Angeles, Calif, USA, June 2000.

[24] L. Zhang, I. Ghosh, and M. Hsiao, "Efficient sequential ATPG for functional RTL circuits," in *Proceedings of IEEE International Test Conference (ITC '03)*, vol. 1, pp. 290–298, Charlotte, NC, USA, September–October 2003.

[25] V. A. Chouliaras and J. Nunez, "Scalar coprocessors for accelerating the G723.1 and G729A speech coders," *IEEE Transactions on Consumer Electronics*, vol. 49, no. 3, pp. 703–710, 2003.

[26] E. Atzori, S. M. Carta, and L. Raffo, "44.6% processing cycles reduction in GSM voice coding by low-power reconfigurable co-processor architecture," *IEE Electronics Letters*, vol. 38, no. 24, pp. 1524–1526, 2002.

[27] J. Hilgenstock, K. Herrmann, J. Otterstedt, D. Niggemeyer, and P. Pirsch, "A video signal processor for MIMD multiprocessing," in *Proceedings of 35th Design Automation Conference (DAC '98)*, pp. 50–55, San Francisco, Calif, USA, June 1998.

[28] R. Garg, C. Y. Chung, D. Kim, and Y. Kim, "Boundary macroblock padding in MPEG-4 video decoding using a graphics coprocessor," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 12, no. 8, pp. 719–723, 2002.

[29] C. N. Hinds, "An enhanced floating point coprocessor for embedded signal processing and graphics applications," in *Proceedings of Conference Record 33rd Asilomar Conference on Signals, Systems, and Computers*, vol. 1, pp. 147–151, Pacific Grove, Calif, USA, October 1999.

[30] J. C. Alves and J. S. Matos, "RVC-a reconfigurable coprocessor for vector processing applications," in *Proceedings of 6th IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '98)*, pp. 258–259, Napa Valley, Calif, USA, April 1998.

[31] T. Bridges, S. W. Kitchel, and R. M. Wehrmeister, "A CPU utilization limit for massively parallel MIMD computers," in *Proceedings of 4th Symposium on the Frontiers of Massively Parallel Computation*, pp. 83–92, McLean, Va, USA, October 1992.

[32] H. Schmit, D. Whelihan, A. Tsai, M. Moe, B. Levine, and R. R. Taylor, "PipeRench: A virtualized programmable datapath in 0.18 micron technology," in *Proceedings of IEEE Custom Integrated Circuits Conference (CICC '02)*, pp. 63–66, Orlando, Fla, USA, May 2002.

[33] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor, "PipeRench: a reconfigurable architecture and compiler," *Computer*, vol. 33, no. 4, pp. 70–77, 2000.

[34] S. C. Goldstein, H. Schmit, M. Moe, et al., "PipeRench: a coprocessor for streaming multimedia acceleration," in *Proceedings of 26th IEEE International Symposium on Computer Architecture (ISCA '99)*, pp. 28–39, Atlanta, Ga, USA, May 1999.

[35] S. Cadambi, J. Weener, S. C. Goldstein, H. Schmit, and D. E. Thomas, "Managing pipeline-reconfigurable FPGAs," in *Proceedings of 6th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '98)*, pp. 55–64, Monterey, Calif, USA, February 1998.

[36] H. Schmit, "Incremental reconfiguration for pipelined applications," in *Proceedings of 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '97)*, pp. 47–55, Napa Valley, Calif, USA, April 1997.

[37] B. A. Levine and H. Schmit, "Efficient application representation for HASTE: hybrid architectures with a single, transformable executable," in *Proceedings of 11th Annual IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '03)*, pp. 101–110, Napa Valley, Calif, USA, April 2003.

[38] C. Ebeling, D. C. Cronquist, and P. Franklin, "RaPiD - reconfigurable pipelined datapath," in *Proceedings of 6th International Workshop on Field-Programmable Logic and Applications (FPL '96)*, pp. 126–135, Darmstadt, Germany, September 1996.

[39] C. Ebeling, D. C. Cronquist, P. Franklin, and C. Fisher, "RaPiD - a configurable computing architecture for compute-intensive applications," Tech. Rep. TR-96-11-03, University of Washington, Department of Computer Science & Engineering, Seattle, Wash, USA, 1996.

[40] C. Ebeling, D. C. Cronquist, P. Franklin, J. Secosky, and S. G. Berg, "Mapping applications to the RaPiD configurable architecture," in *Proceedings of 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '97)*, pp. 106–115, Napa Valley, Calif, USA, April 1997.

[41] D. C. Cronquist, P. Franklin, S. G. Berg, and C. Ebeling, "Specifying and compiling applications for RaPiD," in *Proceedings of 6th IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '98)*, pp. 116–125, Napa Valley, Calif, USA, April 1998.

[42] D. C. Cronquist, C. Fisher, M. Figueroa, P. Franklin, and C. Ebeling, "Architecture design of reconfigurable pipelined datapaths," in *Proceedings of 20th Anniversary Conference on Advanced Research in VLSI*, pp. 23–40, Atlanta, Ga, USA, March 1999.

[43] E. Mirsky and A. DeHon, "MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources," in *Proceedings of 4th IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '96)*, pp. 157–166, Napa Valley, Calif, USA, April 1996.

[44] U. J. Kapasi, W. J. Dally, S. Rixner, J. D. Owens, and B. Khailany, "The imagine stream processor," in *Proceedings of IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 282–288, Freiberg, Germany, September 2002.

[45] B. Khailany, W. J. Dally, U. J. Kapasi, et al., "Imagine: media processing with streams," *IEEE Micro*, vol. 21, no. 2, pp. 35–46, 2001.

[46] J. D. Owens, S. Rixner, U. J. Kapasi, et al., "Media processing applications on the Imagine stream processor," in *Proceedings of IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 295–302, Freiberg, Germany, September 2002.

[47] J. R. Hauser and J. Wawrzynek, "Garp: a MIPS processor with a reconfigurable coprocessor," in *Proceedings of 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '97)*, pp. 12–21, Napa Valley, Calif, USA, April 1997.

[48] T. J. Callahan, J. R. Hauser, and J. Wawrzynek, "The Garp architecture and C compiler," *Computer*, vol. 33, no. 4, pp. 62–69, 2000.

[49] T. Callahan, "Kernel formation in Garpcc," in *Proceedings of 11th Annual IEEE Symposium on FPGAs for Custom Computing*

Machines (FCCM '03), pp. 308–309, Napa Valley, Calif, USA, April 2003.

[50] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao, "The Chimaera reconfigurable functional unit," in *Proceedings of 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '97)*, pp. 87–96, Napa Valley, Calif, USA, April 1997.

[51] S. Hauck, M. M. Hosler, and T. W. Fry, "High-performance carry chains for FPGAs," in *Proceedings of ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '98)*, pp. 223–233, Monterey, Calif, USA, February 1998.

[52] R. Hoare, S. Tung, and K. Werger, "A 64-way SIMD processing architecture on an FPGA," in *Proceedings of 15th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS '03)*, vol. 1, pp. 345–350, Marina del Rey, Calif, USA, November 2003.

[53] S. Dutta, A. Wolfe, W. Wolf, and K. J. O'Connor, "Design issues for very-long-instruction-word VLSI video signal processors," in *Proceedings of IEEE Workshop on VLSI Signal Processing, IX*, pp. 95–104, San Francisco, Calif, USA, October–November 1996.

[54] A. Capitanio, N. Dutt, and A. Nicolau, "Partitioned register files For VLIWs: a preliminary analysis of tradeoffs," in *Proceedings of 25th Annual International Symposium on Microarchitecture (MICRO '92)*, pp. 292–300, Portland, Ore, USA, December 1992.

[55] "Trimaran, An Infrastructure for Research in Instruction-Level Parallelism," 1998, http://www.trimaran.org.

[56] A. K. Jones, R. Hoare, I. S. Kourtev, et al., "A 64-way VLIW/SIMD FPGA architecture and design flow," in *Proceedings of 11th IEEE International Conference on Electronics, Circuits and Systems (ICECS '04)*, pp. 499–502, Tel Aviv, Israel, December 2004.

[57] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communications systems," in *Proceedings of 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '97)*, pp. 330–335, Research Triangle Park, NC, USA, December 1997.

[58] J. Degener and C. Bormann, "GSM 06.10 lossy speech compression library," available on-line: http://kbs.cs.tu-berlin.de/~jutta/toast.html.

[59] G. Golub and C. F. V. Loan, *Matrix Computational*, Johns Hopkins University Press, Baltimore, Md, USA, 1991.

[60] B. Hassibi and H. Vikalo, "On sphere decoding algorithm. I. Expected complexity," submitted to *IEEE Transactions on Signal Processing*, 2003.

[61] B. Hassibi and H. Vikalo, "On sphere decoding algorithm. II. Examples," submitted to *IEEE Transactions on Signal Processing*, 2003.

[62] Y. Chobe, B. Narahari, R. Simha, and W. F. Wong, "Tritanium: augmenting the trimaran compiler infrastructure to support IA64 code generation," in *Proceedings of 1st Annual Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Techniques (EPIC '01)*, pp. 76–79, Austin, Tex, USA, December 2001.

Raymond R. Hoare is an Assistant Professor of Electrical Engineering at the University of Pittsburgh. He received his Bachelor of Engineer degree from Steven's Institute of Technology in 1991. He obtained the Master's degree from the University of Maryland and his Ph.D. from Purdue University in 1994 and 1999, respectively. Dr. Hoare teaches hardware design methodologies at the graduate level, computer organization, and software engineering. His research focus is on high performance parallel architectures. For large parallel systems, his focus is on communication and coordination networks. For systems on a chip, he is focused on parallel processing architectures and design automation for application specific computing. Dr. Hoare is one of the founders, and is the General Chair for the IEEE Workshop on Massively Parallel Processing.

Alex K. Jones received his B.S. in 1998 in physics from the College of William and Mary in Williamsburg, Virginia. He received his M.S. and Ph.D. degrees in 2000 and 2002, respectively, in electrical and computer engineering at Northwestern University. He is currently an Assistant Professor at the University of Pittsburgh in Pittsburgh, Pennsylvania. He was formerly a Research Associate in the Center for Parallel and Distributed Computing and Instructor of electrical and computer engineering at Northwestern University. He is a Walter P. Murphy Fellow of Northwestern University, a distinction he was awarded with twice. Dr. Jones' research interests include compilation techniques for behavioral synthesis, low-power synthesis, embedded systems, and high-performance computing. He is the author of over 30 publications related to high-performance computing and power-aware design automation including a book chapter in *Power Aware Computing* (Kluwer, Boston, Mass, 2002). He is currently an Associate Editor of the International Journal of Computers and Applications. He is also on the Program Committee of the Parallel and Distributed Computing and Systems Conference and the Microelectronic System Engineering Conference.

Dara Kusic is a Masters student at the University of Pittsburgh. Her research interests include parallel processor design, hybrid architectures and computational accelerators. She is a Student Member of the IEEE and the IEEE Computer Society.

Joshua Fazekas is an M.S.E.E. student at the University of Pittsburgh. His research interests include hardware/software codesign, compiler design, and low-power hardware design. Fazekas received a B.S. in computer engineering from the University of Pittsburgh.

John Foster is a Masters student in the Department of Electrical and Computer Engineering, University of Pittsburgh. He received his B.S. degree in computer engineering from University of Maryland, Baltimore County. His research interests include parallel processing compilers and hardware/software codesign.

Shenchih Tung is a Ph.D. candidate in the Department of Electrical and Computer Engineering, University of Pittsburgh. He received his B.S. degree in electrical engineering from National Taiwan Ocean University, Taiwan, in June 1997. He received his M.S. degree in telecommunication from Department of Information Science at the University of Pittsburgh in August 2000. His research interests include parallel computing architecture, MPSoCs, network-on-chip, parallel and distributed computer simulation, and FPGA design. He is a Member of the IEEE and IEEE Computer Society.

Michael McCloud received the B.S. degree in electrical engineering from George Mason University, Fairfax, VA, in 1995, and the M.S. and Ph.D. degrees in electrical engineering from the University of Colorado, Boulder, in 1998 and 2000, respectively. He spent the 2000–2001 academic year as a Visiting Researcher and Lecturer at the University of Colorado. From 2001 to 2003 he was a Staff Engineer at Magis Network, Inc., San Diego, California, where he worked on the physical layer design of indoor wireless modems. He spent the 2004 and 2005 academic years as an Assistant Professor at the University of Pittsburgh. He is currently with TensorComm, Inc., Denver, Colorado, where he works on interference mitigation technologies for wireless communications.