Pipelining Tradeoffs of Massively Parallel SuperCISC Hardware Functions

Colin J. Ihrig, Justin Stander, Alex K. Jones

University of Pittsburgh
Dept. of Electrical and Computer Engineering
Pittsburgh, PA 15261, USA
{cihrig, jstander, akjones}@engr.pitt.edu

Abstract

Parallel processing using multiple processors is a well-established technique to accelerate many different classes of applications. However, as the density of chips increases, another technique to accelerate these applications is the use of application specific hardware processing blocks in parallel within a chip. SuperCISC hardware blocks utilize this method to accelerate scientific, signal, and image processing applications. By applying pipelining methodologies to SuperCISC functions, the effective amount of parallelism already present can be further increased. Automated register placement within a combinational data flow graph (DFG) is governed by the desired maximum operating frequency provided as a parameter to the tool flow, as well as the results of static timing analysis of the circuit. Results presented include the design tradeoffs between increased performance, area, and energy. Additionally, benefits of pipelining compared to hardware replication as a means of achieving further parallelism is studied.

1. Introduction

Pipelining is a common technique used in modern processors to increase performance by overlapping multiple instructions in different phases of execution in an assembly line fashion. While pipelining does not improve the latency of a single instruction, the effects of parallel execution can boost the overall throughput of the processor, while also allowing a higher maximum clock frequency. The effects of pipelining can be seen best when new inputs can be provided at every clock cycle, and all the stages of the pipeline are doing useful work.

1-4244-0910-1/07/$20.00 ©2007 IEEE

Another increasingly popular way of achieving greater speedup is to move performance intensive software code blocks into hardware functions, which can execute many instructions in parallel. These super-complex instruction set computing (SuperCISC) hardware kernels can be tightly coupled with a standard reduced instruction set computing (RISC) processor to form a heterogeneous processor architecture. This heterogeneous architecture, due to its SuperCISC hardware, is capable of massive amounts of instruction level parallelism (ILP) not achievable in similar homogeneous processor architectures.

The goal of the SuperCISC method is to generate highly parallel hardware functions with extremely low latency of execution. These hardware functions are entirely combinational. Control flow dependencies that typically denote cycle boundaries have been converted into data flow dependencies through a technique called *hardware predication*. The resulting hardware functions tradeoff circuit area for performance and reduced energy.

SuperCISC hardware typically dedicates new computational elements for items that can proceed in parallel. For example, a loop may be unrolled and independent loop iterations may occur entirely in parallel. We call this technique *loop iteration replication*.

In this paper, we examine the introduction of pipelining as a form of parallelism in the application and compare pipelining to unrolling and replication. Pipelining allows both resources to be reused, saving area, while allowing these resources to execute concurrently. By combining the width parallelism of the SuperCISC method with the depth parallelism of pipelining the datapath, we can see marked improvements in throughput while still retaining a relative low latency.

The rest of this paper is organized as follows: Section 2 explains the process for generating the SuperCISC hardware from C code, and how these hardware functions extract parallelism. Section 3 shows how the

combinational hardware functions can be pipelined automatically using static timing analysis. Section 4 describes the design tradeoffs that arise from pipelining, such as increases in latency, throughput, chip area, and energy consumption. Section 5 provides comparisons between pipelining and function replication as two ways of increasing throughput. The design tradeoffs of each method are compared in terms of area and energy. Section 6 provides conclusions and discusses future work related to the project.

2. SuperCISC Hardware Functions

A general trend observed in application profiling is that 10% of the code is responsible for 90% of the total execution time [1]. In many scientific, multimedia and signal processing applications, the execution time is dominated by loops within computationally intense functions. Profilers can be used to identify the computational kernels in the software code. These kernels are ideal targets for hardware acceleration. Many algorithms have been moved from software to hardware manually, but automated techniques of doing so are still not always successful. By using an automated C to VHDL tool flow based in the SUIF compiler system, the identified sections of code are converted into a control and data flow graph (CDFG) intermediate representation. The CDFG is subjected to multiple passes of processing in order to generate more efficient SuperCISC hardware. One example of such a pass made on the CDFG is bit-width analysis, which is used generate more area efficient hardware by minimizing bus widths used by functional units.

In [2], a heterogeneous architecture is introduced which consists of a RISC very long instruction word (VLIW) processor with application specific combinational SuperCISC hardware functions, shown in Figure 1. The SuperCISC functions do not contain any storage elements, but are tightly coupled with the VLIW processor through the use of a shared register file. The four-wide VLIW is capable of supporting four instructions executing in parallel. However, due to a lack of instruction level parallelism extracted from software by the compiler, typically seen ILP rarely reaches two [3].

In [4] and [5], the power and performance effects of moving software kernels into SuperCISC hardware functions were studied. Because hardware executes more quickly than software, the overall time required to do the same amount of processing is reduced. This increase in throughput also contributes to a reduction in the amount of energy consumed. By pipelining these hardware functions, additional levels of throughput can be achieved while reducing switching energy. In some of the benchmarks observed, the reduction in switching energy was significant enough to reduce the overall energy consumption of the circuit.

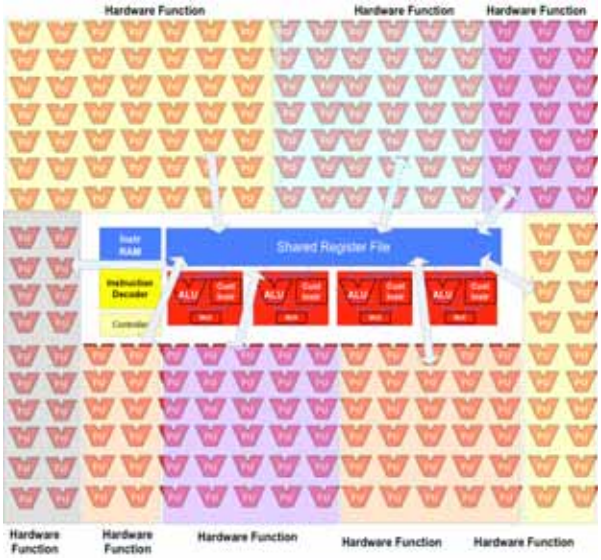

Figure 1. SuperCISC concept with a VLIW processor core and hardware functions surrounding.

2.1 Related Work

Several projects such as PipeRench [6] and HASTE [7] have investigated the mapping of execution kernels into coarse-grained reconfigurable fabrics utilizing arithmetic and logic units (ALUs). The RaPid project [8] presents another coarse-grain, pipelined, configurable architecture. The SuperCISC approach, in contrast, by generating completely application specific hardware, attempts to minimize the amount of area and energy used while sacrificing reconfigurability. The custom hardware lends itself towards a flexible, datapath driven pipelining algorithm. The SuperCISC method also suffers less cycle time waste than coarse-grain fabrics because the pipeline is tailored specifically to the datapath.

Other projects have studied ways to reduce power in high-performance systems without sacrificing speed. In [9] and [10] power and performance tradeoffs are explored using methods such as dual speed pipelines. By using high-speed and low-speed pipelines in conjunction, performance is increased, while area is sacrificed. In contrast, SuperCISC functions use hardware predication to achieve high-performance by expanding the kernel in a single very large dataflow graph. This graph is then rebuilt with pipelining to retain as much latency reduction as possible while simultaneously expanding the parallelism.

A common problem in the domain of hardware-software codesign is partitioning, or determining which parts of the application will be implemented in hardware

and which parts will be implemented in software. The tool flow used in this paper relies on profiling to determine the execution kernels which will be implemented in hardware. Behavioral synthesis techniques can then be used to generate hardware descriptions from high-level languages such as C [11]. High-level synthesis is an increasingly popular technique in hardware design. Mentor Graphics' Catapult C product creates synthesizable hardware descriptions directly from a C/C++ front end [12]. The PACT project from Northwestern University creates power and performance optimized hardware descriptions from C programs [13]. The SPARK project from UC Irvine also converts C code to hardware, while attempting to achieve greater performance by extracting parallelism [14].

In contrast, SuperCISC generates entirely combinational Super Data Flow Graphs (SDFGs) through the use of hardware predication. The remainder of this section will describe the behavioral synthesis techniques applied in our system to the kernels of C code.

2.2 SuperCISC Hardware and Parallelism

Once profiling has identified the execution kernels of an application, the corresponding sections of code can be marked for conversion to hardware. An example of such a code block is shown in Figure 2. A small example is used here instead of a more complex benchmark to ease the readability of the resulting SDFG diagrams in this paper. The tool is able to map arithmetic operations into a DFG representation. Control statements such as *if-then-else* structures are mapped into multiplexers in hardware. In hardware, both the if and the else branches of a control statement are executed simultaneously. The select signal of the multiplexer is driven by the result of the if statement's condition. This removes all control dependencies and replaces them with data dependencies. This conversion process is the core of hardware predication. Intermediate storage values that do not live beyond the end of the hardware function are also eliminated. Figure 3 illustrates the resulting SDFG for the simple example from Figure 2. The SDFG does not include the C variables `i`, `m`, `n`, and `z` as they are only live during the course of the hardware function. The two multiplexers in the graph correspond directly to the two if statements in the code segment.

The amount of parallelism that can be achieved by a hardware function is highly dependent on the nature of the application. In [3], a detailed explanation of the increased parallelism is explored. This subject will be revisited here. Some applications have a control flow based kernel. The ADPCM Encoder and Decoder are examples of control intensive kernels. Other applications have kernels consisting of a large percentage of arithmetic nodes with little to no control flow. The MPEG and JPEG benchmarks all lack any control flow. While control intensive hardware functions achieve increased levels of parallelism relative to software, arithmetic hardware functions benefit the most. The maximum increase in effective parallelism achievable by a control structure is equal to the number of nodes in the larger branch. Although all of the branches will be executed in parallel, only one branch will provide meaningful results, while the others will be discarded. The increase in effective parallelism achieved by strictly arithmetic based kernels is always equal to the total number of nodes. MPEG, the largest of the benchmarks, does not contain any control flow. The effective parallelism achieved by MPEG is 866.

```
1  // Begin Hardware Function
2  z = x + y;
3  m = y << 3;
4  n = m - y;
5
6  if ( n < 0 )
7       n = 0;
8
9  if ( q == 3 )
10      i = z + 5;
11 else
12      i = z - 2;
13
14 j = n * i;
15 // End Hardware Function
```

Figure 2. C language software code for kernel portion of a simple example.

3. Automating Pipelining

In order to introduce registers into a combinational datapath, it is necessary to add a clock signal. The clock signal frequency was a user-defined input to the tool. The clock period will dictate how much combinational logic delay can be placed between any two pipeline registers. In order to know at compile time how much combinational logic will fit into a single pipeline stage, it is necessary to obtain worst-case delay characteristics for each type of functional unit present in the SDFG. With this knowledge, the tool can perform static timing analysis to determine the delay between any two nodes in the graph. For any path through the graph, the worst-case combinational delay is given by Σd_i, where d is the delay associated with a given node, and i spans over each of the nodes in the path. The path with the largest Σd_i value is the critical path and determines the performance limitations of a combinational hardware function. By

applying knowledge of how synthesis tools implement and combine certain constructs within a design, it is possible to achieve slightly more accurate static timing results. For example, a shifter with a constant shift amount input can be synthesized as wires and constants, while a shifter with a variable shift amount cannot. Because of this, a constant shift takes less time to execute in hardware. Table 1 shows the critical path lengths and corresponding maximum frequencies found by performing static timing analysis on the benchmark hardware functions.

Table 1. Critical path length and corresponding maximum frequency of combinational benchmark kernels.

Benchmark	Critical Path Length	Frequency
ADPCM Dec	6.80 ns	147.06 MHz
ADPCM Enc	10.00 ns	100 MHz
IDCT Col	16.60 ns	60.24 MHz
IDCT Row	15.40 ns	64.94 MHz
JPEG	25.95 ns	38.54 MHz
Laplace	7.85 ns	127.39 MHz
MPEG	39.60 ns	25.25 MHz
Synth Filter	15.99 ns	62.54 MHz
Sobel	10.4 ns	96.15 MHz

Once static timing is complete, the process of partitioning the SDFG into multiple pipeline stages begins. The algorithm attempts to place nodes into partitions in an as-soon-as-possible fashion. Each functional unit, having some amount of delay associated with it, will consume a certain portion of the available clock cycle time. The pipelining algorithm attempts to fit as many nodes as possible into as few partitions as possible without violating timing restrictions. In order to take full advantage of a pipelined function, it is desirable to be capable of processing new inputs on each clock cycle. For this reason, another constraint on pipelining is that a node cannot be partitioned until all of its parents have been partitioned first to prevent a possible data dependency violation. It is possible for a child node to be placed in the same partition as all, some, or none of its parents.

Partitioning begins by placing all inputs and constant values into the first partition. Each remaining node, *n*, that has not been partitioned is inspected. Each parent node of *n* that has not been partitioned will be visited and partitioned. Once all of the parent nodes have been partitioned, *n* is partitioned. A node will be placed into one of two possible partitions. The first possible partition is that of its parent node with the highest partition number. The first partition created has the lowest partition number, while the last partition created for the circuit has the highest partition number. If the node will not fit into its parent's partition due to timing constraints, then the node will be placed in the next partition in the pipeline. In cases where parent nodes reside in different partitions, it is necessary to align the inputs using dummy registers, whose only purpose is to delay data arrival by a clock cycle.

Figure 4 shows the pipelined equivalent of the SDFG in Figure 3. The combinational example SDFG has been divided into two pipeline stages. A clock input signal and an optional output register has also been included. All of the inputs and constant values have been added to the first partition. In this simple example, all of the remaining combinational logic is able to be placed into the first partition with the exception of the multiplier, which has a relatively high delay. The number of pipeline stages for each benchmark is shown in Table 2.

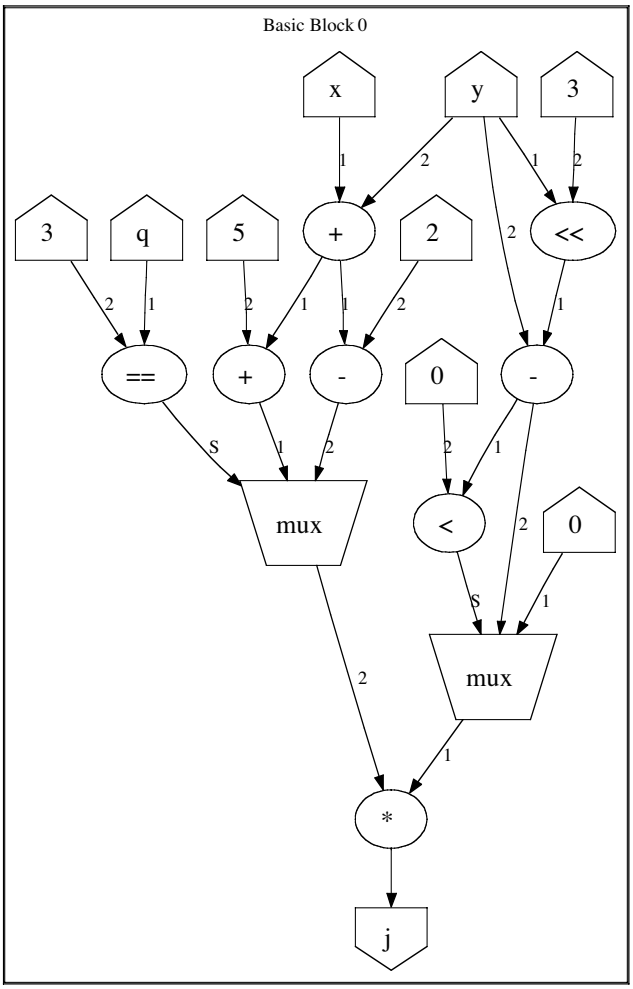

Figure 3. SDFG representation of the simple example.

The clock period of the pipeline can be selected arbitrarily, with the minimum possible period achievable governed by the node(s) in the graph associated with the

largest delay. Register setup and hold times, as well as routing delays must also be taken into consideration in order to ensure that the requested clock period can be met. It should be noted that while the results of static timing are technology dependent, the algorithm is not. Similarly, the results of pipelining are dependent on the results of static timing as well as the requested clock period. The pipelining algorithm itself is also technology independent.

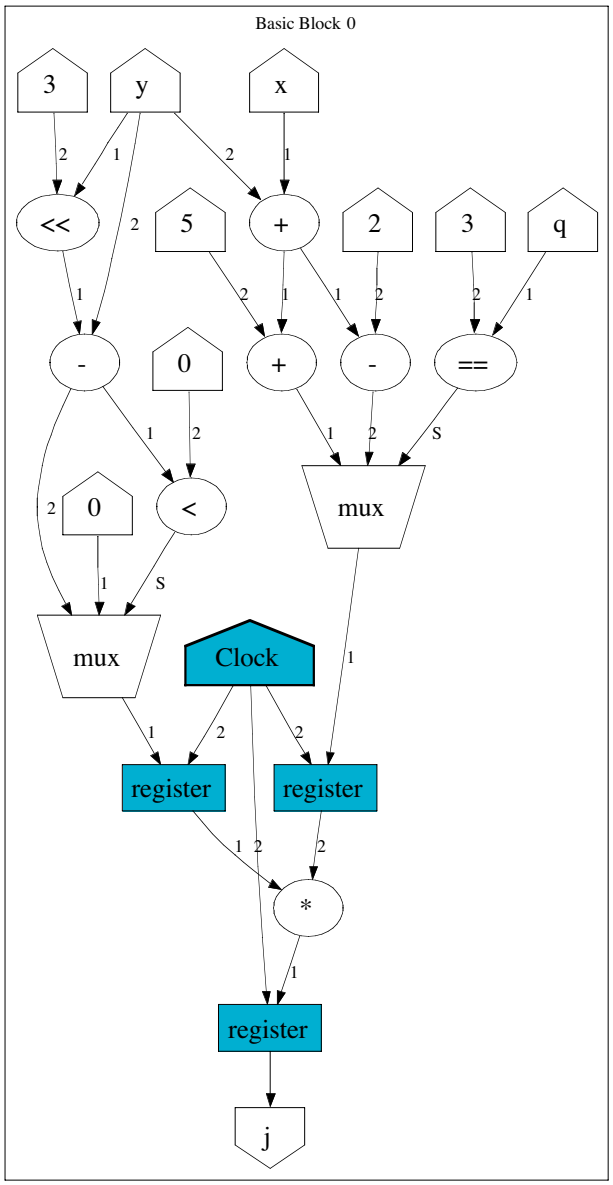

Figure 4. Two-stage pipelined SDFG representation of the simple example.

4. Design Tradeoffs

In this section, we examine the impact of SuperCISC synthesis and pipelining on the parallelism, performance, area, and energy required for the implementation. The results presented here were obtained by applying a pipelining frequency of 200 MHz. The target technology is the 0.16 µm OKI cell-based ASIC technology. Synopsys Design Compiler and PrimePower were used to generate synthesis and power estimations.

Table 2. Number of pipeline stages for hardware functions pipelined at 200 MHz.

Benchmark	Pipeline Stages
ADPCM Dec	2
ADPCM Enc	3
IDCT Col	6
IDCT Row	5
JPEG	9
Laplace	2
MPEG	12
Synth Filter	4
Sobel	3

Part of the speedup achieved by SuperCISC functions comes from *cycle compression* [3]. Cycle compression is the execution of a sequence of arithmetic operations in a fraction of the time that it would take to execute the same operations in software. This is possible because the operations in the software implementation suffer from *cycle fragmentation*. In a processor, the clock cycle length must be long enough to accommodate the delay associated with the critical path. However, simple operation such as logical OR takes only a fraction of the entire cycle time to execute. The remainder of the cycle time is lost to fragmentation. Due to the strictly combinational nature of the SuperCISC hardware, cycle fragmentation is almost completely eliminated. The only remaining fragmentation comes in the final cycle of latency, between the time when the hardware function has completed and the time of the next clock edge. The result is a hardware function that, on average, executes at over 10x the speed of software alone.

SuperCISC hardware functions suffer from two major drawbacks. First, they incur a latency of several processor clock cycles due to their relatively long critical path lengths. In addition, these functions cannot begin processing a new set of input stimuli until the current set has finished and the result stored. This limitation impedes the throughput metric of the hardware kernels.

Pipelining cannot provide a solution to the latency problem associated with the SuperCISC functions. In fact, pipelining can cause an increase in the cycle count due to the reintroduction of cycle fragmentation. The amount of cycle fragmentation introduced through pipelining is dependent on the requested pipeline frequency and the delays associated with the varying types of functional units being pipelined. Figure 5 shows the effects of cycle fragmentation on both the

combinational and pipelined versions of the ADPCM Decoder benchmark. As the frequency increases, additional cycles of latency are incurred. Between 150 and 200 MHz, the frequency increases, while the cycle latency remains unchanged, resulting in the drop in execution time seen in the graph. The effects of higher frequencies cause the pipelined implementation to become fragmented more quickly. The rise in fragmentation causes additional cycles of latency to become necessary, which increases the overall execution time. Figure 6 illustrates the effects of pipelining on the cycle count latency of the benchmark applications. The cycle count, as opposed to the absolute delay, is used to more accurately describe latency because a function's results will only become useful to synchronous hardware on the first clock edge following execution completion.

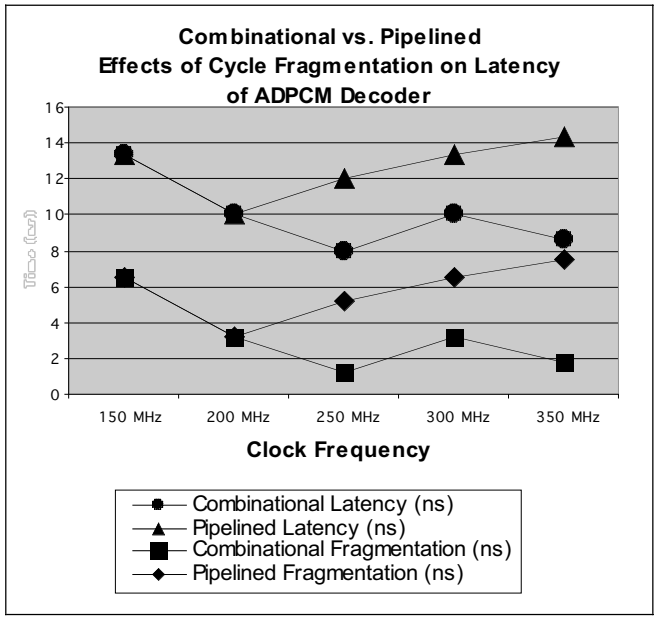

Figure 5. Cycle fragmentation losses incurred in combinational and pipelined versions of ADPCM Decoder at varying frequencies.

While pipelining may degrade the latency of SuperCISC hardware, it can be used to improve the throughput. Combinational implementations of hardware functions can process only a single set of inputs at any given time. This constraint results in a hardware function with a throughput of one result for every number of processor latency cycles associated with the function. By splitting the datapath of a function into two halves with pipeline registers, each half can process different sets of inputs in parallel, with the two results separated by a single cycle. If new input data can be provided to the function every cycle, then the throughput of this implementation is one result per processor cycle.

Pipelining introduces additional registers into a design. The pipeline registers utilize additional on-chip area. A small optimization that can save on both area and energy is the exclusion of any registers storing constant values. Constant values will never change, making storage redundant. The additional hardware also consumes energy. Additionally, the introduction of a clock signal increases the dynamic power of a circuit. Figure 7 displays the performance, area, and energy tradeoffs observed in the pipelined hardware functions versus their combinational equivalents. The performance, on average, increased by 3.3x while the area only increased by 1.4x.

Power is defined as the amount of energy transferred per time unit. Because pipelining allows multiple sets of data to be processed simultaneously, the amount of time needed for the combinational implementation to complete the same amount of work can be significantly longer depending on the length of the critical path. For this reason, power consumption cannot be used to achieve a fair comparison. The amount of energy used is the product of power and time, so it is used to provide a better comparison.

Essentially, the pipelined circuits were high-power, low-energy, while the combinational circuits were low-power, high-energy. The performance gains were also greater than the increase in energy for each case with the exception of the ADPCM Decoder, the smallest benchmark. While the performance and area always increased in the pipelined implementations, the energy did not. This is due to a drop in extraneous combinational switching in the pipelined functions. Figure 8 illustrates the general trend of increased switching energy in combinational hardware functions. Although the overall energy consumption typically went up an average of 1.26x due to the introduction of a clock and registers, the combinational switching energy generally decreased from 57.9% of the overall energy consumption to 35.7%.

5. Kernel Replication vs. Pipelining

Most modern soft core processors have the ability to extend their instruction set with custom instructions utilized in working with co-processors. SuperCISC hardware functions, when implemented as custom instructions, can be tightly coupled with RISC soft core processors to provide a massively parallel processing solution. Many SuperCISC functions can be combined on a single chip to exploit an even higher degree of parallelism. Due to the iterative nature of most vector-based signal processing applications, the utilization of co-processors has been investigated extensively [15]. Replication of hardware functions can be used to unroll time consuming loops and observe greater amounts of speedup. By implementing two instances of the same hardware function, the amount of parallelism is doubled.

Unfortunately, the amount of on-chip area and power consumed is also doubled by replicating the kernel. As the size and complexity of the hardware functions increase, overcoming the area and power usage becomes a more daunting task.

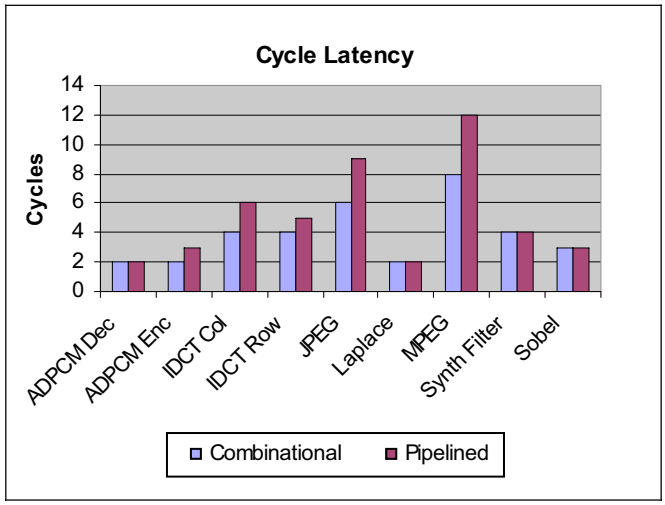

Figure 6. Clock cycle latency comparison of pipelined and combinational versions of SuperCISC functions.

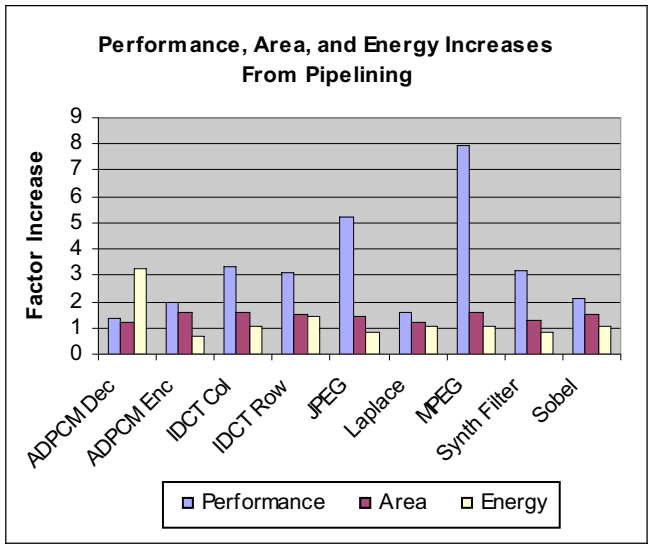

Figure 7. Performance, area, and energy increase for a pipelined hardware function versus its combinational equivalent.

Pipelining provides an efficient, scalable solution to the problems facing replication. As hardware functions are replicated, the area and power increase is linear in the number of replicated instances. By pipelining a single instance of a hardware function, the only area and energy increases are seen from the inclusion of the pipeline registers and clock signal used to drive them. The benefits to this approach can be most easily seen in larger hardware functions. Larger hardware functions typically result in longer pipelines than those of smaller functions. Pipelined functions with *n* pipeline stages approach the performance of functions with a replication factor of *n*. In larger applications, the number of instances needed to achieve this level of full replication places heavy demands on area and energy utilization. Figure 9 shows the increases in area and energy usage with full replication compared to an equivalent pipelined implementation.

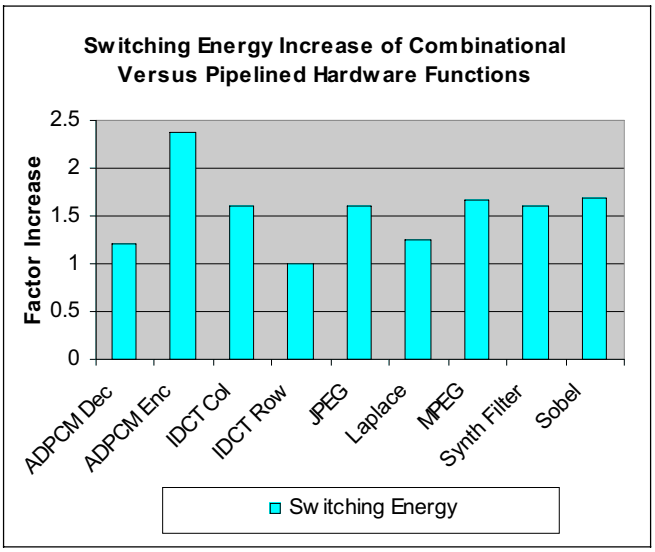

Figure 8. Increase of switching energy in combinational hardware functions versus pipelined equivalents.

6. Conclusions

Levels of instruction level parallelism realized in today's software applications are disappointingly low. Hardware, on the other hand, lends itself towards exploiting parallelism. C to VHDL tool chain technologies are maturing, making hardware design in high-level programming languages increasingly popular. By executing massively parallel hardware functions as custom instructions coupled with standard processors, speedups of over 10x are possible. By pipelining these hardware functions, an average additional performance gain of 3.3x was seen, yielding an increase of 33x over a software only approach. Increasing parallelism even more is possible by replicating hardware functions, while consuming more area and power. Pipelining can be used to achieve the same gains in parallelism and performance while area and energy savings increase with the size of the

replicated function. The largest observed hardware functions saved over 6x the area and over 10x the energy by pipelining instead of replicating.

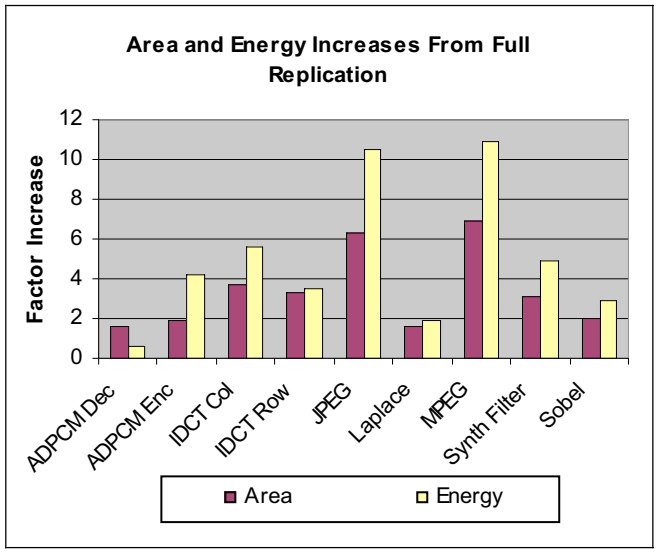

Figure 9. Area and energy increase for a fully replicated hardware function versus its pipelined equivalent.

Future directions of this work seek to automate and optimize the performance, area, and energy tradeoffs associated with electronic design. This is achievable by mixing pipelining and replication techniques based on characteristics specific to an application. Additional work will investigate the presentation of optimization opportunities to the compiler. Possible compiler optimizations exposed include software pipelining and loop unrolling, with and without the presence of loop dependencies.

7. References

[1] R. Hoare, A.K. Jones, D. Kusic, J. Fazekas, J. Foster, S. Tung and M. McCloud, "Rapid VLIW Processor Customization For Signal Processing Applications Using Combinational Hardware Functions," *Int. Symposium on Field Programmable Gate Arrays*, Monterey, CA, Feb. 2005.

[2] A. K. Jones, R. Hoare, D. Kusic, J. Fazekas, and J. Foster, An FPGA-based VLIW Processor with Custom Hardware Execution, *ACM International Symposium on Field-Programmable Gate Arrays (FPGA)* 2005, pp. 107-117.

[3] D. Kusic, R. Hoare, A.K. Jones, J. Fazekas, and J. Foster, "Extracting Speedup From C-code With Poor Instruction-level Parallelism," *19th International Parallel and Distributed Processing Symposium*, Denver, CO, April 2005.

[4] A. K. Jones, R. Hoare, D. Kusic, J. Fazekas, G. Mehta, and J. Foster, A VLIW Processor with Hardware Functions: Increasing Performance While Reducing Power, *IEEE Transactions on Circuits and Systems II*, Vol. 53, No. 11, November 2006, pp. 1250-1254.

[5] A. K. Jones, R. Hoare, D. Kusic, G. Mehta, J. Fazekas, and J. Foster, Reducing Power while Increasing Performance with SuperCISC, *ACM Transactions on Embedded Computing Systems (TECS)* - Vol. 5, No.3, August 2006, pp. 658-686.

[6] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor, "PipeRench: A Reconfigurable Architecture and Compiler" in *IEEE Computer*, Vol.33, No. 4, April 2000.

[7] B. A. Levine, H. Schmit, "Efficient Application Representation for HASTE: Hybrid Architectures with a Single, Transformable Executable." *FCCM* 2003.

[8] C. Ebeling, D. C. Cronquist, P. Franklin, "RaPiD – Reconfigurable Pipelined Datapath", in *The 6th International Workshop on Field-Programmable Logic and Applications*, 1996.

[9] V. Tiwari, D. Singh, S. Rajgopal, G. Mehta, R. Patel,and F. Baez, "Reducing power in high-performance microprocessors," in *Proceedings of Design Automation Conference - DAC*, 1998.

[10] R. Pyreddy and G. Tyson, "Evaluating Design Tradeoffs in Dual Speed Pipelines," in *Workshop on Complexity-Effective Design*, Goteborg, Sweden, June 2001.

[11] D.E. Thomas, J.K. Adams, H. Schmit, "A Model and Methodology for Hardware-Software Codesign," in *IEEE Design & Test of Computers*, September 1993.

[12] S. McCloud, "Catapult C synthesis-based design flow: Speeding implementation and increasing flexibility," Tech. rep., Mentor Graphics, 2004.

[13] 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," Kluwer Academic Publishers, Boston, MA, 2002.

[14] S. Gupta, R. Gupta, N. Dutt, and A. Nicolau, "SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits," Kluwer Academic Publishers, Boston, MA, 2004.

[15] T. Bridges, S.W. Kitchel and R. M. Wehrmeister, "A CPU Utilization Limit for Massively Parallel MIMD Computers," *Fourth Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, October 1992.