A High-Performance Microarchitecture with Hardware-Programmable Functional Units

Rahul Razdan[*+] and Michael D. Smith[*]
[*]Harvard University, Cambridge, MA 02138
[+]Digital Equipment Corporation, Hudson, MA 01742

Abstract

This paper explores a novel way to incorporate hardware-programmable resources into a processor microarchitecture to improve the performance of general-purpose applications. Through a coupling of compile-time analysis routines and hardware synthesis tools, we automatically configure a given set of the hardware-programmable functional units (PFUs) and thus augment the base instruction set architecture so that it better meets the instruction set needs of each application. We refer to this new class of general-purpose computers as PRogrammable Instruction Set Computers (PRISC). Although similar in concept, the PRISC approach differs from dynamically programmable microcode because in PRISC we define entirely-new primitive datapath operations. In this paper, we concentrate on the microarchitectural design of the simplest form of PRISC—a RISC microprocessor with a single PFU that only evaluates combinational functions. We briefly discuss the operating system and the programming language compilation techniques that are needed to successfully build PRISC and, we present performance results from a proof-of-concept study. With the inclusion of a single 32-bit-wide PFU whose hardware cost is less than that of a 1 kilobyte SRAM, our study shows a 22% improvement in processor performance on the SPECint92 benchmarks.

Keywords: programmable logic, general-purpose microarchitectures, automatic instruction set design, compile-time optimization, logic synthesis

1 Introduction

A number of studies have shown that the use of hardware-programmable logic, such as FPGAs, can improve application performance by tailoring hardware paths to match the particular characteristics of the individual application [4,5,6,17]. Overall, the architectures in these studies only work well for special-purpose domains such as logic simulation and large number multiplication. To effectively use hardware-programmable resources in general-purpose environment, we must develop a new approach that is cost-effective, automatic, and applicable to the vast majority of applications.

Our architectural approach to achieve these goals is called PRogrammable Instruction Set Computers (PRISC). To be cost effective, we implement PRISC on top of an existing high-performance processor microarchitecture. For this paper, we use a RISC architecture as our base, though our PRISC techniques are equally applicable to a CISC architecture. PRISC augments the conventional set of RISC instructions with application-specific instructions that are implemented in hardware-programmable functional units (PFUs). These PFUs are carefully added to the microarchitecture so we maintain the benefits of high-performance RISC techniques (e.g. fixed instruction formats) and we minimally impact the processor's cycle time.

MICRO 27- 11/94 San Jose CA USA
© 1994 ACM 0-89791-707-3/94/0011..$3.50

To generate these application-specific PFU instructions in an automated fashion, we have developed compilation routines that analyze the hardware complexity of individual instructions. Using this information, the compiler interacts with sophisticated logic synthesis programs to select sequences of instructions that will execute faster if implemented in PFU hardware. Since the PFU instruction generation process is driven by the specific computations found in each application, our PRISC approach avoids the semantics gap problems of CISC architectures [14]. Furthermore, the complexity of our approach is completely hidden from the user/programmer.

The most general computational model for a PFU is a multi-cycle sequential state machine. Iterative hardware solutions for square-root or transcendental function evaluation are good examples of this class of PFU. The general model however introduces synchronization complexities between the PFU and the other RISC functional units. For this paper, we discuss a simpler model that implements a combinational function of two inputs and one output. The synthesis routines constrain the complexity of this combinational function so that its delay is equal to the delay of the ALU already in the processor datapath. With these two restrictions, a PFU can use the same synchronization mechanisms as the other RISC functional units. We refer to this first implementation of the PRISC architecture as PRISC-1.

PRISC-1 was originally meant as a proof-of-concept vehicle that would allow us to develop the basic PRISC compilation and synthesis environment. To our surprise, the PRISC-1 microarchitecture exhibited noticeable performance benefits not only on the computer-aided design (CAD) applications in the SPECint92 benchmark suite, but on the other applications as well. Even though a PFU is significantly slower than a highly-customized RISC functional unit, we can automatically find opportunities to use a PFU where a typical custom functional unit is not adequate. Our PRISC environment makes the MIPS 1% rule work on a per application basis [18].

The next section summarizes some work related to the use of programmable logic in processor design and the automatic generation of instruction sets. Section 3 describes the microarchitecture of PRISC-1, while Section 4 overviews our PRISC compilation environment and hardware extraction techniques. Section 5 discusses our performance modeling environment and the results obtained from our proof-of-concept experiment. Finally, Section 6 presents conclusions and describes our future work.

2 Related Work

High-level synthesis [10] and automated instruction set generation [13,15,16] are active areas of research in the CAD community, and although the recent work in these areas is relevant to our work, each group is trying to solve slightly different problems. Unlike the work in high-level synthesis which typically attempts to build an application-specific processor automatically, our work adds programmable logic to a general-purpose processor, and it relies on the compiler and run-time system to dynamically reconfigure the programmable logic for each application. Unlike the work in automated instruction set design which systematically analyzes a set of benchmark program to define an entirely-new instruction set for a given microarchitecture, our work simply extends an existing instruction set, and it explores microarchitectures that can effectively adapt to as-yet-unseen applications. Overall, the key aspect of our work is that we produce a complete system where compiler-generated information is used to dynamically reconfigure a relatively small amount of hardware-programmable logic on a per-application basis. We organize this hardware-programmable logic so that it augments the other high-performance techniques found in today's microarchitectures and so that it interacts cleanly with the functionality of today's operating systems.

In many ways, our work is similar to the earlier work in writable microcode stores [1,20,24,29]—each technique dynamically augments the base instruction set with new application-specific instructions to improve application performance. The writable microcode approach creates new instructions by grouping together primitive datapath operations. Performance improves if we can reduce the instruction fetch requirements, use faster data storage, or increase the overlap between operations. However, as Holmer [16] points out, most of these benefits are already obtained by the use of pipelining, multiple issue, and large register files in today's architectures. Our work also creates new instructions by grouping together individual operations in the base instruction set, but since our approach optimizes hardware at a level lower than the existing functional units, we can obtain performance benefits beyond those captured by pipelining and multiple issue techniques. In effect, we pipeline operations at a granularity that is smaller than the existing cycle time. Additionally, only our work addresses the issues involved in dynamically extending the instruction-set architecture of a microprocessor that is used in a multitasking environment.

Previous work in the use of hardware-programmable logic for general-purpose computing has been sparse. Iseli and Sanchez [17] propose a VLIW microarchitecture consisting solely of PFUs. Their processor does not include custom VLSI functional units for typical integer and floating-point operations. Because programmable logic is significantly slower than a custom logic, their prototype had a maximum clock frequency of 5 MHz. This 5 MHz clock rate is significantly below the clock frequencies of today's RISC microprocessors (typically over 60 MHz). In addition, Iseli and Sanchez [17] do not offer any techniques to compile programs from a general-purpose language such as C to their totally programmable environment. Shortcomings such as these make this type of an approach inappropriate for general-purpose microprocessors.

Athanas and Silverman [5] propose an instruction-set augmentation process with general-purpose computing goals that are similar to ours. Like our approach, they describe a compilation process that is coupled with logic synthesis steps. Their compilation process converts entire C functions into programmable hardware. This granularity is much larger than our approach which considers any grouping of instructions as candidates for hardware synthesis. We expect that our more general approach would find greater opportunities for hardware synthesis. Even so, Athanas and Silverman [5] report impressive speedups for a number of specific C routines when run on their PRISM-1 prototype. Overall, there are a number of shortcomings in their initial work that our work attempts to overcome. In particular, their prototype compiler requires some user interaction while our prototype compiler is fully automated, they report performance results only for hardware-optimized routines while we report results for entire applications, and they add programmable logic to a relatively-slow microprocessor (10-MHz M68010) while we experiment with fast cycle times (200 MHz).

In contrast with the sparse work in the general-purpose applications of programmable logic, there has been a great deal of research on programmable-logic solutions which solve domain-specific problems. This work was pioneered by the PAM group in Paris [6]. Their system contains XILINX [31] programmable boards on the I/O bus of a general-purpose workstation. Their approach partitions the computation for a particular problem between the XILINX boards and the workstation processor. The PAM system has shown good results on over ten applications [7], including long integer multiplication [25] and RSA decryption [26]. The SPLASH group from Brown University [4] has mimicked the PAM model and has been successful in solving problems such as text searching, DNA comparison, and edge detection in graphics applications. Unfortunately, these board-based methods incur a high overhead for communicating between the host CPU and the programmable logic board. This significant overhead limits the applicability of this approach to a class of algorithms that have a combination of high computational complexity and low communication overhead.

3 PRISC-1 Microarchitecture

PRISC-1 offers a relatively small amount of hardware-programmable resources—typically 10-times less than that found in existing board-level designs. As Figure 1 shows, we attach the hardware-programmable resources directly to the CPU datapath in the form of a PFU. In general, the implementation of a particular function in a PFU is significantly slower[1] than the implementation of the same function in a highly-customized functional unit. As such, PFUs are added in parallel with the existing functional units so that they augment (not replace or replicate) the existing datapath functionality.

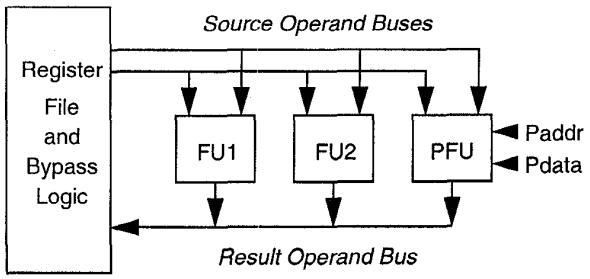

Figure 1: PRISC-1 datapath.

Even though PFUs offer few hardware-programmable resources, these resources reside inside the CPU chip. This design decision minimizes the communication costs (bandwidth and latency) for loading and accessing a PFU. In PRISC-1, PFU data communication is handled just like any other functional unit; each PFU has

1. For example, Lewis [19] reports a factor of three performance difference between programmable circuits and mask-programmed (gate array) circuits.

two input ports where it accepts operands and a single output port where it drives its results. Our hope is that with an efficient communication mechanism, hardware-programmable logic will be useful in a larger class of applications. Each PFU also contains two programming ports; their operation is described in Section 3.2.

The next two subsections discuss the design for a PFU and the extensions to the instruction set architecture needed to program and use this PFU. Section 4 then describes the overall software architecture for PRISC-1.

3.1 PFU design

The design of a PFU is an interesting, non-trivial hardware design problem. For PRISC-1, the primary PFU design constraint is to build a functional unit with a delay that fits within the evaluation phase of our base CPU pipeline. Within this constraint, we must choose a design that maximizes the number of "interesting" functions that can be implemented by the PFU. A function is "interesting" if we can evaluate it faster with the PFU than with the base CPU instructions.

Figure 2 illustrates an example implementation of a PFU for combinational functions (i.e. for PRISC-1). This PFU is comprised of alternating layers of two basic components: interconnection matrices and logic evaluation units. Each possible interconnection point in the interconnection matrix is implemented with a CMOS n-channel transistor that is controlled by a memory cell[2]. By appropriately setting the value in the memory cell, we can connect or disconnect the two lines. Each logic evaluation unit implements a hardware truth table, called a Look-Up Table (LUT). A n-input, 1-output LUT consists of a multiplexer connected to 2^n memory cells (one memory cell per truth table entry). Each memory cell in a PFU is addressable, and in fact, all of the PFU memory cells can be viewed as a large SRAM which is loaded by using the PFU's Paddr and Pdata ports. Programming a PFU to implement a particular function then consists of loading the appropriate values into the interconnection matrix memory cells and the LUT memory cells.

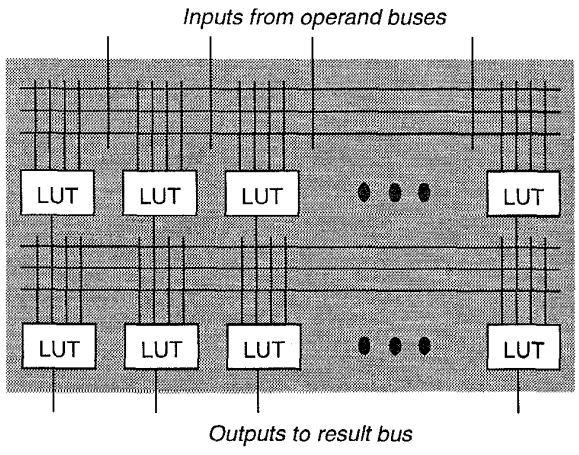

Figure 2: An example of a symmetric, layered PFU. A symmetric PFU implements the same amount of hardware in each bit position.

Because both the interconnection matrix and the logic evaluation units make prodigious use of memory cells, these memory cells dominate the layout cost of our PFU. We have found that a symmetric PFU with 3 alternating layers of interconnect and LUTs requires 30,528 transistors for a 32-bit datapath (61,056 transistors for a 64-bit datapath) where memory cells comprise over 90% of these transistors [22]. Because of the predominance of memory cells, the layout cost of a PFU tracks the layout cost of a SRAM. In comparison, our 30,528-transistor PFU takes considerably less silicon area than a 1 kilobyte SRAM (which requires approximately 50,000 transistors). Current microprocessors easily include over 16 kilobytes of SRAM cache [12], and many have translation lookaside buffers (TLBs) which are larger than a PFU.

As we mentioned earlier, we constrain the latency of a PRISC-1 PFU so that the latency of PFU execution fits within the cycle time of today's high-speed microprocessors. It is straightforward to determine the worst-case delay through the PFU design in Figure 2, and thus by limiting the number of logic levels in our PFU, we can easily bound its delay. Assuming that today's deeply pipelined processors tolerate approximately 15–20 levels of 2-input logic gates per clock cycle, a 3-layer PFU should fit comfortably within a 200 MHz cycle time [22].

The inclusion of a PFU within the datapath places some extra capacitive loading on the operand buses and it increases the size of the multiplexer that feeds the pipeline latch/register at the end of the pipeline evaluation phase. If the delay through the evaluation phase defines the cycle time, then the inclusion of a PFU will increase the cycle time slightly. For other designs where the cycle time is defined by the cache access time or by the branch delay, we can probably add a PFU without affecting the CPU cycle time.

3.2 Instruction Set Extensions

To program and operate PFUs, we define a single new user instruction, the Execute PFU (*expfu*) instruction. Figure 3 presents the format of this instruction in MIPS [18] notation.

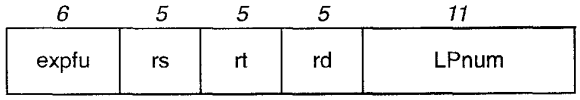

Figure 3: Format of the 32-bit Execute PFU instruction. The *rs* and *rt* fields specify the source operands registers, the *rd* field specifies the destination register, and *LPnum* is number indicating the requested logical PFU function.

The 32-bit *expfu* instruction evaluates a boolean function of two inputs and one output. The compilation/synthesis system assigns a logical-PFU number to each boolean function that it extracts from an application. The *LPnum* field in an *expfu* instruction specifies the particular extracted function to execute; the 11 bits in the LPnum field allow for a maximum of 2048 different PFU programming configurations per application.[3] As explained in Section 4, the programming information for each logical-PFU function is part of the data segment of the application's object file. We use the logical-PFU number to index into the data segment and find the appropriate programming information.

Of course, if we had to program the PFU every time we used it, the latency of a PFU operation would be much greater than the expected value of a single cycle. Thus, we associate an 11-bit register, the *Pnum* register, with each PFU. This Pnum register contains the logical-PFU function currently programmed into the

2. We use the SRAM-fuse technology of programmable logic as the basic fuse primitive in our PFUs since these types of fuses are easy to reprogram.

3. Our current compilation system typically extracts fewer than 200 PFU functions per application.

physical PFU. If the LPnum in the instruction matches the value in the Pnum register, the *expfu* instruction executes normally (and in a single cycle). If there is a mismatch however, an exception is raised, and an exception handler loads the PFU with the correct programming information. Our software is sophisticated enough to determine when it is not beneficial to insert *expfu* instructions, e.g. it is usually a bad idea to insert two *expfu* instructions within a single loop if the hardware contains only a single PFU resource.

The beauty of this approach lies in the fact that a PFU does not add any extra process state that we would need to save on a context switch. By reserving LPnum zero to represent an unprogrammed PFU and by having the hardware[4] clear the Pnum register on an exception or system call, we are guaranteed to force a re-programming of the PFU on its next use. Thus, similar to the handling of TLBs and virtual caches without process ID tags, the cost of a context switch should include a penalty for any PFUs that are re-programmed because of the context switch.

The latency of the *expfu* exception handler depends on the density of the programming memory and the hardware resources allocated for PFU programming. In practice, the PFU programming memory is sparsely populated, typically less than 15% of the bits are asserted. A scheme that relies on a hardware reset mechanism to de-assert all of the PFU memory bits and then only programs the asserted memory locations, would significantly reduce the overall latency to program a PFU. Even if we use this optimization, we still have a range of programming options with widely different hardware and cycle count costs. A simple solution for PFU programming might use implementation-specific load/store instructions in a privileged routine (e.g. an ALPHA PAL routine [11]) to sequentially load the PFU programming memory. A higher performance solution might rely on dedicated programming hardware in combination with a high bandwidth path to memory. For example, if we need to program 20% of the PFU memory bits, the *expfu* exception handler latency under the PAL approach could be as high as 600 cycles, while the high-performance solution could bring this latency below 100 cycles.

4 PRISC-1 Compilation Techniques

Despite the fact that a PFU is not optimized for any particular boolean function, a PFU can improve overall application performance by evaluating several boolean functions with low hardware complexity in a single cycle. These functions were not included as instructions in the base instruction set because they did not provide a significant performance gain across a wide variety of applications. Section 4.1 briefly presents the structure of our PRISC-1 compilation and hardware synthesis system that extracts these application-specific functions and creates *expfu* instructions. Section 4.2 then describes the key analysis step in our system that keeps the compile time reasonable even though we aggressively search for groups of instructions to implement in a PFU. Section 4.3 presents our current routines for extracting *expfu* instructions, and Section 4.4 discusses some additional techniques that can improve the system effectiveness.

4.1 Software Architecture

Figure 4 shows the overall structure of a PRISC-1 compilation system. The left side of this figure is similar to any high-level language (HLL) compilation system. An application in a HLL is parsed, optimized, and translated into target machine instructions; these instructions are then assembled and scheduled to produce a binary executable. Our compilation environment uses profile information, from performance analysis tools such as *pixie* [27], to improve the results of the optimization and instruction scheduling passes.

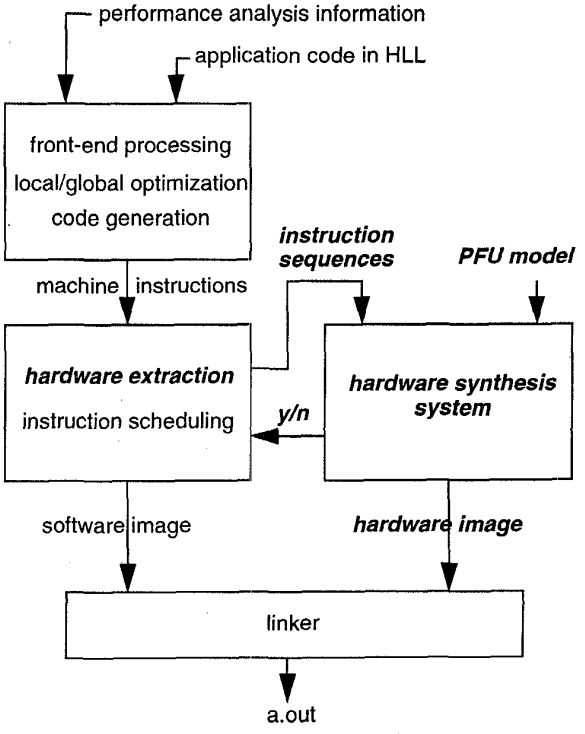

Figure 4: Major passes in a PRISC-1 compilation system. Our hardware extraction routine sends sequences of instructions to a hardware synthesis system which then generates a corresponding PFU programming image. The synthesis system also indicates to the extraction routine whether the resulting image is larger than the physical PFU.

Unlike conventional compilers however, our PRISC-1 compiler inserts a step after code generation, called hardware extraction, that identifies sets of sequential instructions which we could potentially implement with a PFU. Each instruction in an identified set is mapped into boolean operations, and the entire group of boolean operations is given to a hardware synthesis package. The logic synthesis routines take the input combinational function and output a netlist of LUTs. During this step, logic minimization algorithms reduce the number of LUTs and interconnect resources that are used by the input function. Finally, a placement and routing routine is run to determine if the LUT netlist fits in the resources offered by the physical PFU. The result of the place-and-route step is fed back to the hardware extraction routines so that the compiler can automatically reduce the input function if its requirements exceed the physical PFU resources.[5]

Once we have produced the appropriate hardware and software images, all images are linked together in a straightforward manner.

4. Alternatively, the operating system could selectively clear this bit if some system calls did not use the PFU resources.

5. A detailed discussion of the algorithms used for logic synthesis, LUT minimization, and LUT placement-and-route is beyond the scope of this paper. Briefly, we have augmented standard algorithms [8,9,30] to the task of PFU synthesis. In general, the standard algorithms have a worst-case performance behavior which is exponential, but since our extracted functions are combinational and quite small, the existing algorithms, augmented with some simplifications, can quickly synthesize the input functions.

The hardware images (the PFU programming information) simply occupy part of the data segment of the resulting *a.out*. Razdan [22] describes a scheme for maintaining binary compatibility across a family of PRISC machines.

Hardware extraction uses profile information to determine which instructions are executed often enough to justify the expense of programming the PFU. In actuality, this routine does not require profile information for correct operation; without profile information (as with most aggressive compile-time optimizations), the hardware extraction routine is simply more conservative in its selection of optimizations.

4.2 Function-Width Analysis

Even with our modifications to the hardware synthesis routines, it is still relatively expensive to check if an instruction sequence will fit in the physical PFU resources. Consequently, we developed an analysis step, called function-width analysis, that quickly separates instructions into two classes: those that may benefit from PFU conversion and those that definitely will not. This analysis step is based on the observation that a PFU is less efficient than a custom functional unit, i.e. a PFU is unable to evaluate a dense boolean function as fast as a custom functional unit. Since the density of a boolean function is related to the number of literals (input variables) in the function, we can quickly eliminate any instruction whose boolean function requires a large number of input literals. For example, a bitwise AND requires only two input literals per output bit and thus is an ideal candidate for implementation in a PFU. Similarly, a byte-wide ADD requires at most 16 input literals for the most-significant output bit and thus is another excellent candidate. A word-wide (32-bit-wide) ADD, on the other hand, is not a good candidate for implementation in a PFU. Even though a byte-wide ADD and a full-word ADD have the same software costs, they have vastly different hardware costs.

Our routine for function-width analysis performs an iterative algorithm that is very similar to those used in dataflow calculations [2]. The algorithm uses a ternary algebra and goes as follows. The output variable of each unmarked instruction is initialized to X for every bit position. A combination of forward and backward traversals is then made over the control flow graph to reduce the number of X bits. Forward traversals evaluate each instruction and check to see if the evaluation changes the output bit vector. For example, an unsigned byte load zeros all but the lower 8 bits of the result. Backward traversals indicate unnecessary bit calculations. For example, if a variable was stored to memory using a byte store instruction and this variable was not used elsewhere, the instruction that generated the store input need only generate the lowest 8 bits of information. The algorithm ends when no bits change during an iteration. Given the bit values for all of the variables in the application, we heuristically calculate the hardware complexity of the individual instructions and mark every operation that can be easily implemented in a PFU as PFU-LOGIC candidate. For a RISC instruction set, typically only memory operations, floating-point operations, wide adds, multiplies, divides, and variable-length shifts are not marked as PFU-LOGIC.

4.3 Hardware Extraction

Once the compiler has marked all of the potential PFU-LOGIC instructions in an application, it is ready to select sequences of these instructions for conversion to *expfu* instructions. Though we considered many different ways to select instruction sequences, our current hardware extraction routine follows a simple bottom-up greedy approach. Basically, this approach starts with a PFU-LOGIC instruction and then walks backward (against the flow of control in the control flow graph) as far as possible. The backward walk terminates when the next instruction is not a PFU-LOGIC instruction or when inclusion of the next instruction would produce a function requiring more than two source operands or more than one result. If the corresponding boolean function for this maximal[6] instruction sequence does not fit within the PFU resources, our extraction routine simply prunes an instruction at a time from the top (beginning) of the instruction sequence.

As a first example of the operation of our extraction routine, Figure 5 illustrates two sample code sequences extracted automatically from the *espresso* benchmark [28]. Each example is simply a sequence of data-dependent PFU-LOGIC instructions that a PFU can evaluate in a single cycle. If we were to code these same sequences in MIPS R2000 instructions [18], each sequence would require multiple instructions and thus multiple cycles to execute. For reference in the results section, we refer to this optimization as a *PFU-expression* optimization.

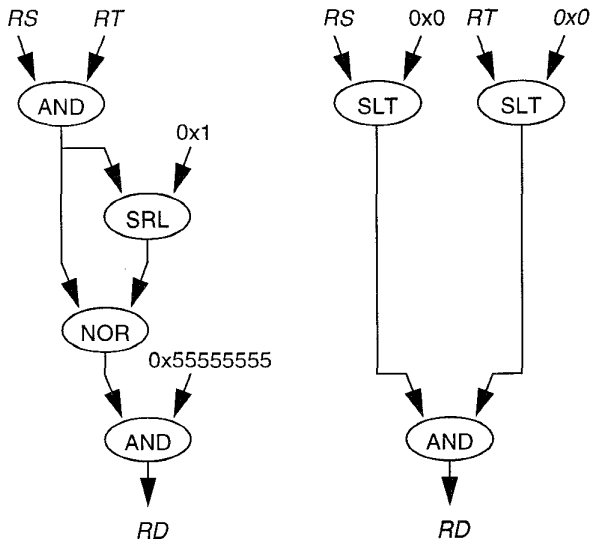

Figure 5: Examples of a PFU-expression optimization. A PFU can evaluate either of these sequences of boolean instructions in a single cycle.

A slightly less obvious type of data flow optimization involves the conversion of table lookups (referred to as *PFU-table-lookup*). Table lookups are used, for instance, when logic expressions become too complex and too inefficient to evaluate directly with the processor's instruction set. If our extraction algorithm can recognize a constant array as a data structure that represents a truth table, we can minimize the table and represent it in a functional form that is evaluated efficiently by a PFU.[7] For illustration, Figure 6 shows a truth table, the MIPS R2000 code, and the PFU logic required for the evaluation of a ternary NAND gate. The MIPS R2000 instruction set requires at least four instructions to evaluate this or any other two-input ternary gate through table lookup techniques. As the figure shows however, a PFU can easily evaluate this ternary gate in a single cycle. In fact, a single 4-input

6. Our simple bottom-up greedy algorithm does not attempt to increase the size of an instruction sequence by rearranging the order of instructions. An obvious next step would be to integrate the hardware extraction routine with the instruction scheduling routines.

7. Of course, as with any type of programmer-applied optimization, life would be much easier if we did not have to undo their optimization to apply ours. Section 4.4 discusses other issues related to this topic.

LUT can evaluate any ternary 2-input gate. Razdan [22] describes a number of other example PFU-table-lookup optimizations.

Truth table

A	B	O
00	XX	00
01	00	00
01	X1	10
01	1X	10
10	00	00
10	01	10
10	10	01
10	11	11
11	00	00
11	X1	11
11	1X	11

R2000 code

```
sllr1, A, 2
orr1, B, r1
addr1, T, r1
ldbZ, 0(r1)
```

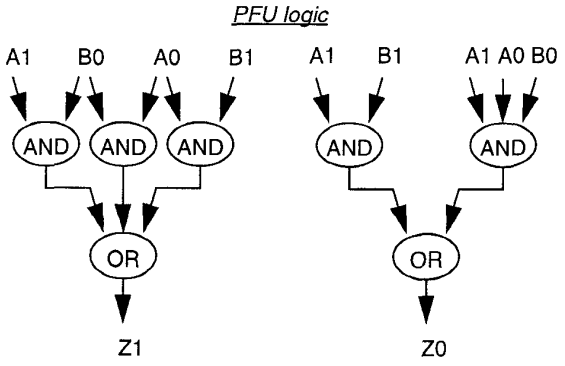

Figure 6: Example of a PFU-table-lookup optimization. The truth table evaluates a ternary NAND gate where 00 is an illegal state, 01 is a logic zero, 10 is a logic one, and 11 is a logic X. In the MIPS R2000 code, *T* is the base address of a fully-decoded table.

In addition to simple sequences of data-dependent PFU-LOGIC instructions, our hardware extraction routine also recognizes opportunities to optimize the control flow of an application. One such opportunity is the *PFU-predication* optimization which transforms an IF-THEN-ELSE-structured portion of a control flow graph (CFG) into a set of boolean equations. In effect, this optimization provides support for a limited form of predicated execution (see Mahlke et al. [21] for more information on hardware and software support for predicated execution). To apply this optimization, the candidate portion of the CFG must adhere to the following characteristics: there must be one and only one entry point into this portion of the CFG; there must be one and only one exit point from the selected portion; every block excluding the entry and exit block must contain only PFU-LOGIC instructions; and the selected portion cannot contain any backward CFG edges (i.e. loops).

Once we have met these constraints, the process of conversion proceeds in three basic steps: predicate assignment, boolean transformation, and boolean minimization. Assignment of predicates to basic blocks is a well understood problem [3]. Once we have calculated the basic block predicates, the compiler transforms the individual PFU-LOGIC instructions in each basic block to include the effects of the predicate. Given a basic block predicate P and an assignment of the form $Z = A\ op\ B$ where op is any of the PFU-LOGIC operations, the boolean transformational rule is (expressed with C-style logical operators):

$$Z_{new} = ((A\ op\ B)\ \&\ (P)^n)\ |\ (Z_{old}\ \&\ (!P)^n)$$

The variables Z_{old} and Z_{new} are the values of the output variable before the assignment and immediately after the assignment respectively. The $()^n$ function takes a boolean bit and generates a n-bit vector containing n copies of this bit. After transformation, we can execute all of the operations independent of the actual control flow. Only those operations with an asserted predicate will affect the value of the result. Figure 7 illustrates the result of a PFU-predication optimization that translates an example code segment into a set of boolean equations.

Example C code

```
If (c == 'b') n = 8;
else if (c == 's') n = 16;
else if (c == 'w') n = 32;
else n = 0;
```

Boolean PFU equations

$$n3 = ((\overline{c4}\ c5)(c6\ \overline{c7}))\ ((\overline{c2}\ \overline{c3})(\overline{c0}\ c1))$$

$$n4 = ((c4\ c5)(c6\ \overline{c7}))\ ((\overline{c2}\ \overline{c3})(c0\ c1))$$

$$n5 = ((c4\ c5)(c6\ \overline{c7}))\ ((c2\ \overline{c3})(c0\ c1))$$

Figure 7: Example of a PFU-predication optimization. The PFU output bits for n that are not shown are tied to logic 0.

The obvious benefit of the PFU-predication optimization is that it reduces the execution time of a portion of a CFG to a single cycle. Another important benefit of this optimization is that it eliminates conditional branches from the instruction stream, and conditional branches are a major impediment to higher performance through instruction scheduling. Unfortunately, IF-THEN-ELSE structures that use non-PFU-LOGIC instructions or that have multiple exit points cannot benefit from the PFU-predication optimization. Even so, we are able to use PFUs in another way that is also beneficial in reducing both execution time and the branch impediments to code motion. This new technique, called the *PFU-jump* optimization, attempts to convert a set of IF-THEN-ELSE statements into a switch statement. This optimization is based on the observation that a significant portion of a program's branches only branch a short distance [14]. Thus a branch can be thought of as a sparse

boolean function—a PFU can evaluate the switch condition and generate the appropriate target address.

In order to use the PFU-jump optimization, a subset of the CFG must: have one and only one start basic block; contain only PFU-LOGIC instructions for the conditional expressions; not contain any backward edges (loops). Figure 8 shows an example of the PFU-jump optimization from the *massive_counts* routine in the *espresso* benchmark. Since the code uses load and store instructions (non-PFU-LOGIG instructions) while incrementing the array locations in the IF bodies, we cannot use the PFU-predicate optimization. The conditional evaluation however requires only PFU-LOGIC instructions, and thus we can replace the three conditional branches with a single dynamic jump whose target address is generated by an *expfu* instruction.[8] Since there are 2^3 different possible execution paths through the code in Figure 8, the compiler optimizes for 8 different target instruction sequences.

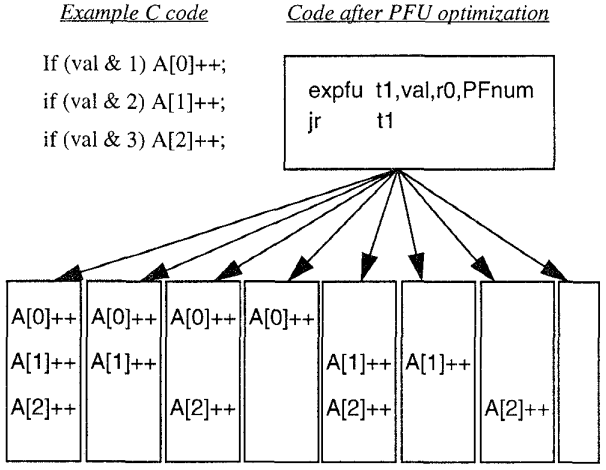

Figure 8: Example of the PFU-jump optimization from the *espresso* benchmark. The register t1 is a temporary register and r0 contains the integer value 0.

Figure 8 also shows the possible negative ramifications of the PFU-jump optimization. First, there is a significant increase in the code size, and this increase may degrade instruction cache performance. Second, this technique forces a premature evaluation of all conditional expressions in the CFG graph. This premature evaluation can degrade performance in CFGs where the shortest path is executed with the highest probability (though this is tempered by the fast evaluation of the switch condition by the PFU). We use branch probability data to determine when a PFU-jump optimization will most-likely improve performance.

The last significant restriction on the use of a PFU to optimize control flow is that neither of the previous two techniques can contain a loop. However, both of these optimizations interact well with loop unrolling techniques [14]. Razdan [22] describes how we extend a simple loop unrolling algorithm to take advantage of a PFU resource (called the *PFU-loop* optimization).

8. To actually generate the target address in the *expfu* instruction requires our system to function also as a link-time optimization; we cannot know the final target addresses until that time.

4.4 Other Optimization Opportunities

The effectiveness of the PFU optimizations described in the previous subsection is limited by the compiler's ability to determine the precise functionality of a set of instructions. For example, the function implemented by a truth-table lookup is easily converted into a *expfu* instruction if the compiler is able to identify this programmer-applied optimization. Similarly, there are a number of character manipulation and string-to-number conversion routines in the C run-time library that could definitely benefit from a PFU resource; however, the hand-tuning of these routines for individual instruction set architectures has made it very difficult to reconstruct automatically their logical intent. In general, we have found that it is possible to structure a wide variety of applications so that they achieve dramatic increases in performance from a PFU resource [23].

5 Performance Modeling and Results

A complete analysis of our PRISC-1 approach would involve the detailed design of a PFU-augmented datapath and the development of the full compilation/synthesis system described in Section 4.1. Before investing heavily in these two activities, we constructed a proof-of-concept system that gave us a rough first estimate the potential performance benefit of a PRISC-1 computer. Section 5.1 describes our performance modeling environment while Section 5.2 presents the results of our simulation study.

5.1 Performance Model

We base our proof-of-concept study on a mythical 200MHz MIPS R2000 datapath that has been augmented with a single PRISC-1 PFU. The datapath microarchitecture and the PFU design match the descriptions given in Section 3. In particular, the PFU is a symmetric, layered PFU designed to meet the requirements of a 200MHz cycle time (i.e. a 1994 technology).

Since we do not have any real hardware, we did not need to develop a complete compilation environment. Our current software system contains implementations of the hardware extraction and function-width analysis algorithms described in Section 4, but unlike the Section 4.1 description of our ideal compilation system, the input to the current routines is a MIPS object file and the output is a change in the cycle count of each object file basic block. Using an object file as input does limit the effectiveness of our extraction routines due to a lack of complete type information. For example, our extraction routines can use a PFU to eliminate the need for temporary registers in an expression evaluation, but if these temporary registers have been spilled to main memory due to register congestion in the original object file, our current algorithms cannot optimize them away—resulting in pessimistic performance results. Similarly, a lack of type information, such as the usage of an enumerated type, greatly limits the effectiveness of our function-width analysis step and thus limits the number of candidate PFU-LOGIC operations. Finally, to keep from duplicating existing compiler functionality, our current software does not perform any compile-time optimizations such as procedure inlining or instruction scheduling. These and other compile-time optimizations could increase the applicability of our various PFU transformations.

As described in Section 4.1, we integrate our analysis and extraction system with routines for hardware synthesis. Again, since we do not have any real hardware, we perform all of the hardware synthesis except LUT placement and routing. To determine if the extracted function will fit in the physical PFU resources, our system implements the following simple rule: if the maximum depth of the gate level network is less than 6 levels, the PFU function is allowed to replace the software code. This estimate of PFU programmability can lead to optimistic results for 5-level networks

which could not fit in our PFU resources, and pessimistic results for 6- or 7-level networks which could have fit into the physical PFU.

We generated the performance numbers in the next subsection by combining basic block cycle counts from our compilation system with basic block execution counts from the pixie profiling tool [27]. Like all performance modeling strategies, this particular strategy has several shortcomings. Since our performance numbers are based on a basic block profile and not on a trace, it is impossible for us to know how often we really need to program a PFU. Currently though, our software system adds only a single *expfu* instruction per loop and only to loops without procedure calls; thus we can easily assume a worst-case scenario for our performance results. Specifically, we assume that a PFU takes 500 cycles to program and that we must re-program each PFU every time we enter its enclosing loop. In other words, we assume that the PFU is never programmed correctly when we enter a loop containing an *expfu* instruction.

Our proof-of-concept system models CPU performance only—memory system penalties are ignored. Similarly, the profile data is for the application execution only—operating system issues and performance are ignored. Because of all of the previous limitations (software and performance modeling), one should view the results as a lower bound on potential decrease in application CPU cycles.

5.2 Performance Results

Since one of our original goals was to develop an approach that is applicable to a wide variety of applications, we selected the SPECint92 benchmark suite [28] as a first cut at a set of diverse applications. We performed all of our experiments on a DECstation 5000/240 using the MIPS C compiler (V2.10). For each benchmark, Table 1 lists the number of times we invoked each of the hardware extraction optimizations. Our software system would apply an individual optimization only if the profile information indicated that the optimization would increase application performance by at least 0.1%. The compilation/synthesis time for the PFU optimizations was typically measured in a few single-digit minutes. Table 2 lists the performance gain obtained on each of the SPECint92 applications. We calculate the performance gain by dividing the number of cycles taken before PFU optimization by the number taken after PFU optimization. The benchmarks shown in Tables 1 and 2 are *compress* (CPS), *eqntott* (EQN), *espresso* (EXP), *gcc* (GCC), *li* (LI), and *sc* (SC).

Optimization	CPS	EQN	EXP	GCC	LI	SC
PFU-expression	9	0	48	13	4	12
PFU-table-lookup	0	0	0	0	0	0
PFU-predication	0	1	0	13	0	0
PFU-jump	10	0	47	103	0	35
PFU-loop	0	3	0	4	0	0
TOTAL	19	4	95	133	4	47

Table 1: Static PFU optimization instances in SPECint92.

	CPS	EQN	EXP	GCC	LI	SC
Speedup	1.15	1.91	1.16	1.10	1.06	1.12

Table 2: Cycle count speedup for a PRISC-1 microarchitecture with a single PFU resource. The speedup for each application is an arithmetic average (as defined by SPEC) of all of the data sets for that application.

Our system found many instances of the PFU-expression and PFU-jump optimizations in four of the six benchmarks. The relatively sparse number of PFU optimization instances found in and low performance improvement of *li* is due to the large number of short procedure calls in the interpreter loop. As Table 2 shows, *eqntott* exhibits an excellent speedup even though it has very few static PFU optimization instances. This significant speedup is due to a single PFU optimization in the *cmppt* routine. This routine accounts for over 85% of the application's cycles so any cycle count decreases in this routine greatly reduce the overall cycle count. The basic data type in *cmppt* is a 16-bit integer, and we have seen a 213% improvement in performance by changing this data type from short (16-bit integer) to char (8-bit integer)! The PFU-table-lookup optimization was never invoked on any of these benchmarks because constant arrays are not declared as constant in application source code. Unfortunately, modifying the application source code with constant qualifiers did not improve this situation because the MIPS C compiler does not retain the read-only nature of constant information in the object file. Finally, it should be noted that the number of PFUs generated by our current system (less than 200 functions per application) does not even approach the *expfu* instruction format limit of 2047 logical-PFU numbers.

For the SPECint92 benchmarks, the performance gain from a single PFU seems significant in comparison with other general-purpose architectural alternatives. For example, consider the addition of more on-chip cache memory. Many of today's commercial microprocessors contain at least 8 kilobytes of on-chip instruction and data cache [12]. Doubling the size of the instruction cache (to 16 kilobytes) only decreases the average instruction cache miss rate by an average of 2% for the SPECint92 benchmarks. Under fairly optimistic conditions (i.e. a CPI of execution equal to 1.0 and a 25 cycle miss penalty), this doubling of the instruction cache provides an average performance improvement of approximately 15%, but at a hardware cost which is eight times that of a PFU.[9]

6 Conclusions and Future Work

This paper has described a novel microarchitecture and compilation/synthesis system that automatically exploits hardware-programmable resources to improve the performance of general-purpose applications. This paper also presented encouraging results from a proof-of-concept experiment that has shown respectable performance gains (22% on the SPECint92 benchmark suite) with a very modest hardware investment (a single combinational PFU). Based on these encouraging results, we have begun to port our PRISC-1 hardware extraction routines to a general-purpose compiler, and in the future, we hope to develop a detailed hardware model of a PRISC-1 datapath.

With a more aggressive compilation environment, we will be able to explore the impact of our techniques on superscalar processors.

9. On a program like *eqntott*, we get a large benefit from the addition of a PFU and nearly no benefit from increasing the instruction cache. At the other end of the spectrum, a program like *gcc* gets a large benefit from doubling the instruction cache, but currently only a small benefit from our PFU. Still in this case, we found that the benefits are fairly equivalent when we added only 1KB of instruction cache.

Our PFU optimizations often reduce register pressure (by eliminating temporary variables), increase the size of basic blocks, or eliminate conditional branches (through predication). All of these side effects have the potential to improve benefit of a superscalar design. Furthermore, we foresee excellent opportunities for synergistic interactions between our hardware extraction algorithms and existing global instruction scheduling algorithms.

7 Acknowledgments

During the research, the assistance of four individuals was invaluable. We would like to thank Bill Grundman for his insightful discussions on custom CMOS implementation techniques and the ramification of these techniques on PFU microarchitecture design. Also, we would like to thank Mark Firstenberg and Ed McLellan for the detailed information they provided on the microarchitectures of two recent high performance VAX and ALPHA architecture implementations. Finally, Steven Morris suggested the PAL approach for programming the PFU.

Digital Equipment Corporation provided funding for Rahul Razdan's graduate work. Mike Smith was funded in part by a NSF Young Investigator Award.

8 References

[1] A. Abd-alla and D. Karlgaard. Heuristic Synthesis of Microprogrammed Computer Architecture. *IEEE Transactions on Computers*, C-23(8):802–807, Aug. 1974.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[3] J. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. *Proc. 10th ACM Symp. on Principles of Programming Languages*, Jan. 1983.

[4] J. Arnold et al. The Splash 2 Processor and Applications. *Proc. Int. Conf. on Computer Design*, Oct. 1993.

[5] P. Athanas and H. Silverman. Processor Reconfiguration Through Instruction-set Metamorphosis. *IEEE Computer*, 26(3):11–18, Mar. 1993.

[6] P. Bertin, D. Roncin, and J. Vuillemin. Introduction to Programmable Active Memories. *Systolic Array Processors*, J. McCanny et al. eds., Prentice Hall, 1989.

[7] P. Bertin, D. Roncin, and J. Vuillemin. Programmable Active Memories: A Performance Assessment. Lecture Notes in Computer Science, Springer Verlag, *Parallel Architectures and Their Efficient Use*, Paderborn 678:119-130, Nov. 1992.

[8] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. MIS: a Multiple-Level Logic Optimization System. *IEEE Transactions on CAD*, CAD-6(6):1062–1081, Nov. 1987.

[9] S. Brown et al. *Field-Programmable Gate Arrays*, Kluwer Academic Pub. 1992.

[10] R. Camposano and W. Wolf, editors. *High-Level VLSI Synthesis*, Kluwer Academic Publishers, 1991.

[11] Digital Equipment Corp. *Alpha Architecture Handbook*, 1992.

[12] D. Dobberpuhl et al. A 200-MHz 64-bit Dual-issue CMOS Microprocessor. *Proc. Int. Solid State Circuits Conf.*, Feb. 1992.

[13] F. Haney. Using a Computer to Design Computer Instruction Sets. Ph.D. thesis, Carnegie-Mellon Univ., Pittsburgh, PA, 1968.

[14] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*, Morgan Kaufman, 1990.

[15] B. Holmer and A. Despain. Viewing Instruction Set Design as an Optimization Problem. *Proc. of 24th Int. Symp. on Microarchitecture*, Nov. 1991.

[16] B. Holmer. Automatic Design of Computer Instruction Sets. Ph.D. thesis, U. of California, Berkeley, CA, 1993.

[17] C. Iseli and E. Sanchez. Beyond Superscalar Using FPGAs. *Proc. Int. Conf. on Computer Design*, Oct. 1993.

[18] G. Kane. *MIPS RISC Architecture*, Prentice-Hall, 1989.

[19] D. Lewis, M. van Ierseel, and D. Wong. A Field Programmable Accelerator for Compiled-Code Applications. *Proc. Int. Conf. on Computer Design*, Oct. 1993.

[20] P. Liu and F. Mowle. Techniques of Program Execution with a Writable Control Memory. *IEEE Transactions on Computers*, C-27(9):816–827, Sept. 1978.

[21] S. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. *Proc. 25th Annual Intl. Symp. on Microarchitecture*, Dec. 1992.

[22] R. Razdan. PRISC: Programmable Reduced Instruction Set Computers. Ph.D. thesis, Harvard University, Cambridge, MA, 1994. Also available as Center for Research in Computing Technology Tech. Rep. TR-14-94, Div. of Applied Sciences, Harvard Univ., Jun. 1994.

[23] R. Razdan, K. Brace, and M. Smith. PRISC Software Acceleration Techniques. *Proc. Int. Conf. on Computer Design*, Oct. 1994.

[24] T. Rauscher and A. Agrawala. Dynamic Problem-oriented Redefinition of Computer Architecture via Microprogramming. *IEEE Transactions on Computers*, C-27(11):1006–1014, Nov. 1978.

[25] M. Shand, P. Bertin, and J. Vuillemin. Hardware Speedups in Long Integer Multiplication. *Computer Architecture News*, 19(1):106, Jan. 1991.

[26] M. Shand and J. Vuillemin. Fast Implementation of RSA Cryptography. *Proc. 11th Symp. on Computer Arithmetic*, 1993.

[27] M. Smith. Tracing with pixie. Computer Systems Lab. Tech. Rep. CSL-TR-91-497, Stanford Univ., Nov. 1991.

[28] Standard Performance Evaluation Corporation (SPEC) Newsletter, Volume 4, Issue 1, Mar. 1992.

[29] J. Stockenberg and A. van Dam. Vertical Migration for Performance Enhancement in Layered Hardware/Firmware/Software Systems. *Computer*, 11(5):35–50, May 1978.

[30] Thomas et al. *Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench*, Kluwer Academic Pub., 1990.

[31] Xilinx Corporation. *Programmable Gate Array Book*, 1989.