

# Processor Reconfiguration Through Instruction Set Metamorphosis: Compiler and Architecture

Peter M. Athanas

Virginia Polytechnic Institute and State University

Bradley Department of Electrical Engineering

340 Whittemore Hall

Blacksburg, Virginia 24060

athanas@vtvm1.cc.vt.edu

Harvey F. Silverman

Brown University

Laboratory for Engineering Man/Machine Systems

Division of Engineering

Providence, Rhode Island 02912

hfs@lems.brown.edu

## Abstract

*Many computationally-intensive tasks spend nearly all of their execution time within a small fraction of the executable code. A new hardware/software system, called PRISM, is presented which improves the performance of many of these computationally intensive tasks by utilizing information extracted at compile-time to synthesize new operations which augment the functionality of a core processor. By integrating adaptation into a general-purpose computer, one can not only reap the performance benefits of application-specific processors, but also retain the general-purpose nature by accommodating a wide variety of tasks. Newly synthesized operations are targeted to RAM-based logic devices which provide a mechanism for fast processor reconfiguration. A proof-of-concept system called PRISM-I, consisting of a specialized C configuration compiler and a reconfigurable hardware platform is presented. Compilation and performance results are provided which confirm the concept viability, and demonstrate significant speed-up over conventional general-purpose architectures.*

**Keywords:** *Adaptive Architectures, Reconfigurable Instruction Sets, Performance Improvements, General-Purpose Computers, Logic Synthesis*

# I Introduction

General-purpose computers are designed with the primary goal of providing acceptable performance on a wide variety of tasks rather than providing high performance on specific tasks. The performance of these machines ultimately depends upon how well the capabilities of the processing platform match the computational characteristics of the applications. If an application requires more computational power than that which can be achieved by a general-purpose platform, one is often driven to an *application-specific* computer architecture in which the fundamental capabilities of the machine are designed for a particular class of algorithms. Tasks suited for a particular application-specific machine perform well; however, the execution performance of tasks outside of the targeted class is usually poor.

One characteristic common to nearly all computationally intensive applications is that they tend to spend most of their execution time within a small portion of the executable code [1]. Efforts to improve the performance of such tasks are best spent on these small, frequently-accessed portions instead of other sparsely-accessed sections. Substantial performance gains can be achieved by allowing the configuration and fundamental operations of a processor to *adapt* to the portions of the program that consume the most execution time. In doing so, segments of the processing platform reconfigure to add new capabilities to accommodate individual applications. Such an architecture would reap the performance benefits of application-specific architectures by tuning the architecture to individual tasks, yet would retain a general-purpose nature by being able to change dynamically to match the character of a wide diversity of tasks.

In this paper, the architecture and compiler components of a general-purpose computing platform called PRISM are presented. PRISM – an acronym for *Processor Reconfiguration through Instruction Set Metamorphosis* – is intended to improve the performance of many computationally intensive tasks by using information extracted from a program during compilation to synthesize new operations that augment the functionality of a processor. Motivation for using adaptation in a general-purpose machine is presented in Section II, along with a discussion of previous work. A proof-of-concept system, called PRISM-I, which consists of a special compiler, called a *configuration compiler*, and a hardware platform, is examined in Section III. Section IV provides experimental results, demonstrating the benefits of the PRISM concept.

## II Adaptation - Previous Work and Motivation

The idea of incorporating some means of adaptation into a computer has been around for almost as long as the digital computer itself. In 1960, Estrin [2] proposed a machine architecture

consisting of a fixed, general-purpose core appended with an inventory of high-speed *substructures*. The fixed portion of the computer centered on a “minimum vocabulary and machine characteristics” common to many applications. A rack of substructures, or application-specific computational elements, provided the means of tuning the fixed core towards application-specific tasks. Unfortunately, the substructure library could offer only a finite number of alternatives from which to choose, and a high-level language compiler intending to make use of these substructures required a great deal of complexity.

Later, in the 1960s and 1970s, many large machines were built with writable *control stores*. Substantial microcoding and decoding was required to implement the large, complex instruction sets for these machines. A writable control store offered a way of utilizing the faster access time of RAM structures, and also allowed easy microcode maintenance of machines in the field. Rauscher and Agrawala introduced a method [3] of task adaptation in machines with writable control stores where new machine microcode would be generated at compile-time from information derived from the user’s program. The compiler proposed in [3] defined a new set of instructions better suited for the task, and generated microprograms that would interpret the new instructions. By the late 1980s, memory technology advanced to the point where entire programs could be compiled into what was essentially “microcode” resident in the machine’s fast store (RISC machines). This rendered many of these large microcoded machines obsolete.

The feasibility and utility of providing some means of adaptation in modern, general-purpose machines has been demonstrated (for example, see [4, 5]). In these papers, speed-up was achieved by essentially replacing portions of applications that were previously executed in software with custom-designed hardware structures that could do the same computation in a fraction of the time. In order to keep the processing platforms general-purpose, RAM-based *field programmable gate arrays* (FPGAs) were used as the mechanism for containing the augmented instructions. These devices are functionally equivalent to medium-density gate-arrays, but, in addition, can be reprogrammed many times [6]. Logic functionality of each FPGA is determined by the output of RAM cells distributed throughout the device, which in turn can be modified under software control. In the cases above, a conventional processor was augmented with one or more FPGAs connected to the processor bus as a coprocessor. Despite the relative simplicity of the processing platforms in all of these, claims of 10- to 1000-fold improvement have been reported by adding only a few instructions to the general-purpose machine. More specifically, the authors in [5] claimed a couple of orders-of-magnitude improvement for some image processing tasks, and in [4], the authors claimed a 2700-time throughput increase over a VAX 11/785 by adding the ability to compare large strings!

There were major barriers that prevented this early work from taking root in mainstream computing. First, the programmer was expected to know where the computationally inten-

sive portions existed within the program. This may not have been an overwhelming problem since many programmers have a good idea of what portions of their code dominate execution time. Then, once these *candidates* were identified, the programmer was assumed to have the hardware-design expertise to define and synthesize hardware structures for these new operations. Furthermore, in order to specify these new structures, the programmer was forced to leave the programming environment and use another language (either a hardware-description language or schematic-entry system) to describe the new structures. To make an adaptive system viable, the *identification* and *synthesis* operations need to be merged within the compilation process, and should be fully automated.

## II.1 An Adaptive Architecture and Compiler

By selecting (or designing) an appropriate programming language, and by providing a suitable hardware platform, the identification and synthesis of enhanced operations can be accomplished automatically during program compilation. As a consequence, the same language used to specify software functionality is also used to specify hardware structures. Therefore, it is the responsibility of the compiler, not the programmer, to decide which portions of the application are to be synthesized. A performance gain is achieved when the time required to evaluate a function on the newly adapted hardware is less than the time required to compute the same operation in software. Also, to insure a benefit throughout the lifetime of the program executable, the processing overhead imposed by this architecture (such as the added compilation time, and reconfigurable resource initialization time) must not exceed the total savings achieved by these methods<sup>1</sup>.

The procedure for automatically synthesizing candidate functions is similar to behavioral silicon compilation [7, 8]. This process involves the transformation of a high-level behavioral description of a task into a structural, and finally into an efficient physical description that is functionally equivalent. The compilation of a high-level-language specification into an efficient hardware implementation is a difficult problem. One reason is because the programmer is limited to the fundamental operations inherent in the high-level language to convey the concepts of an algorithm. All functions, whether simple or complicated, must be expressed as a combination of a limited set of boolean and arithmetic operations provided by the high-level language. However, not all functions are easily expressed in these simple operations. For example, consider the function `Hamming` written in `C` illustrated in Figure 1. This function, as defined here, accepts two unsigned integers and returns a number from `[0..16]` equal to

---

<sup>1</sup>The costs of compilation and initialization may be moot in many applications that execute frequently or possess real-time execution constraints.

---

```

short
Hamming( l1, l2 )
unsigned short l1, l2;
/*****/
/* Returns the Hamming distance of l1 and l2. */
/* The Hamming distance is a metric determined */
/* by the sum of the absolute bitwise difference */
/* of the two operands. */
/*****/
{
    register short i, /* loop counter. */
xor, /* intermediate XOR. */
result; /* final return value.*/

    result = 0;
xor = l1 ^ l2;

    for (i=0; i < sizeof( short); i++ ) {
        result = ( xor & 1 ) + result;
xor = xor >> 1;
    }
    return( result );
}

```

---

Figure 1: The Hamming metric written in C.

their Hamming distance. There is no primitive in C to perform the Hamming metric; thus, when expressing this in the functional set of C as in Figure 1, it becomes an expensive operation (as compared to, say, integer addition)<sup>2</sup>. An application that depends heavily on such a function would obviously benefit if the underlying processor supported this function, and the high-level-language compiler was capable of utilizing it.

Manufacturers of general-purpose microprocessors generally do not include instructions such as Hamming functions in their repertoire, and the reasoning is clear. An instruction such as this has such a rare occurrence in a “general-purpose” instruction mix that the allocation of area on silicon for such functions is not justified. Instead, manufacturers strive to create

---

<sup>2</sup>There are a number of different ways to express this function – depending on several factors, some ways are perhaps more computationally efficient than others. Independent of how it is expressed, the most efficient method of execution is direct hardware evaluation.

Figure 2: An overview of the PRISM configuration compiler and the processing platform. As shown here by the dotted lines, identified portions of the input specification are transformed into equivalent hardware structures.

instruction sets that are *complete* in the sense that instructions not implemented in the set can at least be synthesized by a sequence of included instructions.

In support of the above argument, suppose a `Hamming` instruction was included in a particular microprocessor’s repertoire. To utilize this in a high-level language such as `C`, the high-level-language compiler would be required to recognize the function call, as in Figure 1, written in all its infinite permutations, and compile it into a single instruction. While this compiler operation may not be impossible, it certainly would not be cost-effective relative to both compiler execution time and compiler development time – just for the sake of a very rare instruction.

In this paper, the problem is approached in reverse. Instead of statistically determining the native capabilities of a processor *a priori* from “general-purpose” instruction-mix statistics and fixing them into silicon, one determines – *at compile time* – a set of operations that best suits an application. Once identified, an attempt is made to synthesize these operations for subsequent execution.

In its simplest form, the adaptive-processing platform can be constructed from a conventional microprocessor augmented appropriately with an array of FPGAs; however, to reap the full benefits of this architecture, the reconfigurable resource should be integrated closely with the core of the CPU to maximize the data bandwidth between the two. Unquestionably the most difficult component to develop in this system is the specialized compiler; the configuration compiler is made of a collection of complex components, and will be examined in the next section.

### III The PRISM-I Prototype System

A *configuration compiler* is a special compiler that accepts a program as input, and produces both a *hardware image* and a *software image*. The hardware image consists of physical specifications that are used to program a *reconfigurable platform*. The software image, similar in function to an executable image produced by a conventional compiler, consists of machine code ready for execution on a processor along with code that integrates the newly synthesized element(s) into the execution instruction stream. The relationship between these components is illustrated in Figure 2. The newly-generated hardware augments the capabilities of a

processor so that subsequent *calls* to the function are evaluated as if the function was a native capability of the host microprocessor. PRISM-I is a proof-of-concept system that consists of a *configuration compiler* and *reconfigurable hardware platform*, and has been developed to demonstrate the viability of incorporating adaptation in a general-purpose machine. Information regarding the operation of this compiler can be found in the sidebar titled *The PRISM-I Configuration Compiler*.

This architecture has assumed that the processing platform consists of a reconfigurable resource integrated closely with a central processing core. The core processor is responsible for the execution of the software images and coordinating the execution of synthesized operations. One must realize that reconfigurable media will always have slower propagation times and lower effective gate densities than an ASIC counterpart. This is because the reconfigurability aspect of these devices adds to signal propagation delay (either by loading or longer critical paths), and consumes significant device area. Therefore, for optimum execution performance, a balance must exist between what is to be executed in software, and what is to be executed in hardware.

The above suggests that data must pass between the central processor and the reconfigurable media. To execute a synthesized structure, three steps are required:

1. Input arguments must pass from the processor to the reconfigurable medium. The transfer time is governed by the bandwidth of the bus interconnecting the two components.
2. The reconfigurable medium executes the operation. In non-sequential structures, this would be the time it takes the inputs to propagate to the output(s).
3. The output(s) of the operation are returned to the processor. State information, if generated, may stay within the reconfigurable resource.

A gain in performance is achieved only if the sum of these three times is less than the average time needed to evaluate a given function in software. If need be, these steps can be easily pipelined, and can operate concurrently with processor activities.

The PRISM-I hardware platform was our first attempt at a reconfigurable platform, and was intentionally simple to demonstrate the utility of this architecture. It consisted of an existing processor board – an **Armstrong** processing node [9] – which was based upon a 10 MHz M68010 processor, and a second board which consisted primarily of four Xilinx 3090 FPGAs; these provide an excellent means of enabling the system to reconfigure in less than one second under software control. The two boards were interconnected by a 16-bit bus. Synthesized operations were evaluated by the processor explicitly moving the function input arguments to the appropriate FPGA bank, and then (immediately) moving the computed result back

to the processor board. This whole operation was accomplished with no wait-states, yet still required between 48 to 72 processor clock cycles to complete. Despite the gross inefficiencies in data movement, the prototype system exhibited surprisingly good performance, which will be shown in the next section.

## IV Results

Table 1 summarizes the compilation and run-time performance for several example functions that may be encountered in a number of applications. While most of these functions are identifiable by the reader, useful target functions that would benefit common applications are likely to be more obscure. In these experiments, a SUN-3 optimizing C compiler was used to generate the M68010 code for trials without PRISM-I, and also used to generate the software image for trials utilizing PRISM-I. The speed-up achieved by using PRISM-I versus conventional software evaluation of the listed functions is given in the last column of Table 1. From this table, one can see that a call to the synthesized `Hamming` function will require 1/24th of the time to execute on the PRISM-I system than if it were evaluated entirely in software on the host processor.

It is interesting to note that the function `Bitrev`, a common bit reversal routine used in FFT algorithms, requires a `for`-loop and many logical and shifting operations when written in C (as written in this case). The compiled configuration of this function requires *no logic* but merely a reassignment of the input pins to the outputs. Note that the compile times for most of these examples is under 15 minutes (using a Sun SPARC IPC workstation). The short compilation time was an important design issue to insure the viability of the PRISM concept. The actual performance benefit for applications using these functions is dependent upon how frequently the synthesized structures are referenced by the application. For example, the function `LogicEv` performs the core computation used in a digital circuit discrete-event simulator. If a digital circuit simulator application spent 90% of its execution time calling and executing `LogicEv` in software, then with PRISM-I, the same application would perform 6.67 times faster<sup>3</sup>.

One should note that the PRISM-I platform was hindered in its performance because of the relatively slow interface between the processor core and the reconfigurable media. Modern microprocessors, which are capable of much higher bus transfer rates, can transfer data to/from the reconfigurable devices in significantly less time than that of the prototype system. In planned work, this problem (among others) is curtailed by reducing the number of cycles

---

<sup>3</sup>The speed-up factor in this case is derived by an application of Amdahl's Law [1]:  $6.67 = \frac{18}{0.9+18-0.9 \cdot 18}$ .

required to access the reconfigurable media by a factor of more than 20. Because of this, an additional order-of-magnitude improvement is expected.

Since PRISM-I is a *proof-of-concept* system, certain functions that were not paramount to the operation of the compiler, and have been demonstrated and documented elsewhere, may not have been incorporated into this version of the compiler. At the present time, the input source code to the configuration compiler is constrained by some major limitations:

- State and global variables are currently not supported within the boundary of candidate functions, and must be passed through function arguments; however, local automatic variables may be declared freely.
- Limitations on the proof-of-concept system require the sum of the sizes of the input arguments to be 32 bits or fewer. Likewise, the size of the return value is limited to 32 bits.
- The exit condition for `for`-loops must be independent of the input arguments.
- Floating-point types and operations are currently not supported.
- The synthesized structure will evaluate the function directly in a single cycle. No sequential devices or feedback paths are currently used in synthesized structures.
- Not all `C` constructs have been implemented. These include the popular `do-while` and `switch-case` constructs.

These limitations, among others, offer a substantial challenge for future research in the development of a full-scale compiler.

## V Conclusion

In this paper, an architecture and high-level-language compiler has been presented which together has the potential of significantly improving the execution performance of many applications. This is achieved by allowing the configuration and fundamental operations of a core processing system to *adapt* to the computationally-intensive portions of a targeted application. The viability of having compilers generate hardware configurations in conjunction with executable code means that the user can take advantage of the performance increase offered by application-specific hardware without having to become a hardware designer. The newly synthesized operations are targeted to RAM-based logic devices which provide a mechanism for fast processor reconfiguration.

<i>Function Name</i>	<i>Description (input bytes / output bytes)</i>	<i>Compilation Time (mins)</i>	<i>% Utilization of a XC3090 FPGA</i>	<i>Speed-up Factor</i>
Hamming(x,y)	Calculates the hamming metric. (4/2)	6	38%	24
Bitrev(x)	Bit-reversal function. (4/4)	2	0%	26
Neuron(x,y)	Cascadable 4-input N-Net function. (4/4)	12	52%	12
MultAccm(x,y)	Multiply/accumulate function. (4/4)	11	58%	2.9
LogicEv(x)	Logic simulation engine function. (4/4)	12	40%	18
ECC(x,y)	Error correction coder/decoder. (3/2)	6	14%	24
Find_first_1(x)	Find first '1' in input. (4/1)	3	11%	42
Piecewise(x)	5-section piecewise linear seg. (4/4)	24	77%	5.1
ALog2(x)	Computes base-2 A*log( x ). (4/4)	16	74%	54

Table 1: Compilation and performance results of various functions from the PRISM-I compiler running on a Sun SPARC IPC workstation. *Speed-up* factors represent the speed improvement of executing on a 10 MHz M68010 based **Armstrong** node with PRISM-I relative to the node without PRISM-I. Compilation times do not include target place-and-route times.

The prototype system utilizes commercially available Xilinx FPGAs to facilitate reconfiguration. These are excellent devices for demonstrating PRISM principles; however an FPGA device designed specifically to exploit features of this architecture (such as support for shadow configurations, a fixed input/output structure, and faster reconfiguration time) may be essential for supporting conventional operating system concepts, like context switching, debugging, and resource sharing.

A first implementation of a PRISM system, developed at Brown University's Laboratory for Engineering Man/Machine Systems (LEMS), is complete and operational. A full-scale version of the configuration compiler and hardware platform is currently under development.

## References

- [1] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.
- [2] G. Estrin. Organization of computer systems: The fixed-plus variable structure computer. In *Proceedings of the Western Joint Computer Conference*, pages 33–40, 1960.
- [3] T. G. Rauscher and A. K. Agrawala. Dynamic problem-oriented redefinition of computer architecture via microprogramming. *IEEE Transactions on Computers*, C-27(11):1006–1014, November 1978.
- [4] M. Gokhale, B. Holmes, A. Kopser, D. Kunze, D. Lopresti, S. Lucas, R. Minnich, and P. Olsen. Building and using a highly parallel programmable logic array. *IEEE Computer*, 24(1):81–89, January 1991.
- [5] P. Bertin, D. Roncin, and J. Vullemin. Introduction to programmable active memories. *Digital Equipment Corporation Paris Research Laboratory*, June 1989.
- [6] Xilinx Inc. San Jose, California. *The Programmable Gate Arrays Users Book*, 1991.
- [7] D. Gajski. *Silicon Compilation*. Addison-Wesley, Reading, Massachusetts, 1988.
- [8] T. Tanaka. HARP: FORTRAN to silicon. *IEEE Transactions on Computer-Aided Design*, 8(6):649–660, June 1989.
- [9] J. T. Rayfield and H. F. Silverman. System and application software for the Armstrong multi-processor. *IEEE Computer*, 21(6):38–52, June 1988.