# An Architecture Framework for Introducing Predicated Execution into Embedded Microprocessors

Daniel A. Connors[1], Jean-Michel Puiatti[2], David I. August[1], Kevin M. Crozier[1], and Wen-mei W. Hwu[1]

[1] Department of Electrical and Computer Engineering
The Coordinated Science Laboratory
University of Illinois, Urbana, Illinois (USA) 61801
{dconnors, august, crozier, hwu}@crhc.uiuc.edu
[2] Logic Systems Laboratory (DI-LSL)
Swiss Federal Institute of Technology Lausanne, CH-1015 Lausanne, Switzerland
puiatti@lslsun.epfl.ch

**Abstract.** Growing demand for high performance in embedded systems is creating new opportunities for Instruction-Level Parallelism (ILP) techniques that are traditionally used in high performance systems. Predicated execution, an important ILP technique, can be used to improve branch handling, reduce frequently mispredicted branches, and expose multiple execution paths to hardware resources. However, there is a major tradeoff in the design of the instruction set, the addition of a predicate operand for all instructions. We propose a new architecture framework for introducing predicated execution to embedded designs. Experimental results show a 10% performance improvement and a code reduction of 25% over a traditionally predicated architecture.

## 1 Introduction

Growing demand for high performance in embedded computing systems is creating new opportunities for Instruction-Level Parallelism (ILP) techniques that are traditionally used in high performance systems. In several ways, the needs of embedded computing differ from those of more traditional general purpose systems. Embedded systems have more stringent constraints on cost [2] that lead to the design of limited-sized instruction caches and physical memories. The limited nature of these instruction memory resources is more pronounced by current technological developments in embedded systems. In order to meet numerous requirements for embedded system features and functionality, compilers and high-level languages have been employed in ways to manage the size and complexity of system design. Unfortunately, this can increase program code size over previously used traditional methods of hand-coding programs. Thus, compiler technology not only has a large effect in enhancing the performance of these processors, but also in affecting the instruction memory utilization and

code size. Although classic code optimizations decrease the number of executed instructions, superscalar optimization, inline expansion, loop unrolling, and superblock formation [3] often increase the execution performance at the cost of increasing the overall code size.

Current embedded processor research and development illustrate different strategies for dealing with memory size issues. The traditional strategy for reducing code size focuses on reducing the instruction encoding size in the design of the Instruction Set Architecture (ISA). For example, embedded processor such as M-Core, Thumb, Tiny RISC, and future ARM-10 designs [8] use this technique. A comparison of the cache performance in [1] shows that denser instruction sets have significantly lower miss rates for small caches, but that advantage disappears for larger caches.

Predicated execution is an emerging ILP technique that requires several changes to existing ISAs, which can affect program code size. Predicated execution is the conditional execution of an instruction based on the value of a Boolean source operand, referred to as the predicate of the instruction. Predicated execution, can be used to improve branch handling, reduce frequently mispredicted branches, and expose multiple execution paths to hardware resources. Although the performance benefits of full predicated execution are high, there is a major tradeoff in the design of the instruction set, namely the addition of a predicate source operand for all instructions.

We propose a new framework for introducing predication into embedded processors. The first contribution of this paper is to present the effect of predicated execution on program code size. This study concludes that the growth in binary size when adding predicate source operands to every instruction is wasteful since only 28% of instructions are predicated after applying aggressive predicate formation [7] and optimization techniques. The second, and the more important contribution of this paper is to propose a new instruction issue mechanism that supports predicated and non-predicated versions of instructions. Experimental results of the new predicated architecture model achieves an average 10% performance improvement over a traditionally predicated architecture and reduces the memory requirements of highly optimized code by 25%

## 2 Background and Motivation

### 2.1 Predication Background

Predicated execution allows conditional execution of instructions based upon a computed condition and may be supported by several different architectural models [6]. Each model must support a method of expressing the condition and a method for the condition to affect instruction execution. Full predication supports this using new instruction set and microarchitecture extensions.

The full predication model consists of four components: a predicate register file for holding 1-bit predicate values, an additional source operand for each instruction to specify a predicate for instruction execution, a conditional-execution

stage to nullify instructions, and a set of predicate defining instructions for generating conditions. The values in the predicate register file are associated with each instruction through the use of an additional source operand, or predicate operand. This operand specifies which predicate register will determine whether the instruction should execute. A predicate register value of 1, or true, indicates the instruction is executed; a value of 0, or false, indicates the instruction is suppressed. An unconditional instruction is designated by a predicate register that is always true. The architectural support for predicated execution can be found in the HPL PlayDoh Architecture Specification [4].

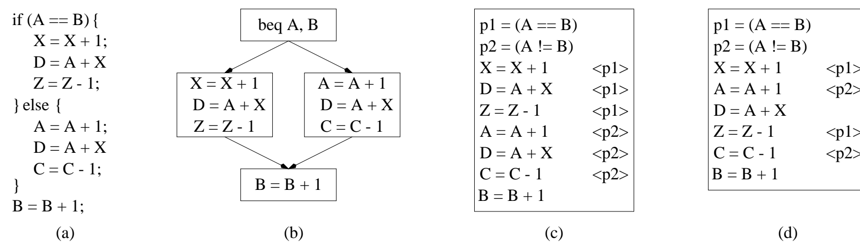


**Fig. 1.** A simple if-then-else C code construct (a), unpredicated code (b), predicated code (c), and optimized predicated code (d).

Predication support allows the compiler to use an *if-conversion* algorithm to convert conditional branches into predicate defining instructions, and instructions along alternative paths of each branch into predicated instructions [9]. Figure 1 demonstrates the limitation of the traditional control flow graph when applied to predicated code. A simple if-then-else construct is shown in Figure 1(a). The code generated for this segment without predication is shown in Figure 1(b). Here the control flow graph clearly shows that one and only one side of the if-statement may execute. The predicated code control flow graph is shown in Figure 1(c). In this case all the code falls into one basic block because there is no possibility of branching until the end of the set of instructions.

The most notable modification of predication to the instruction set encoding format is the addition of the predicate operand source for every instruction. The predicate operand increases the instruction size and has significant effects on overall program code size. One model [10] proposes a new set of predicate guarding instructions that would reduce the drawback of existing methods of specifying predicated execution through the use of predicate mask-setting instructions. Although the mechanism is useful in reducing the predicate operand overhead, the general mechanism constrains several aspects of predicated execution and dramatically alters the instruction issue logic of microprocessors.

### 2.2 Motivation

**Predication Performance.** There are two major benefits associated with applying if-conversion. First, a compiler can eliminate problematic branches from

the program. In doing so, all the associated overhead with these branches is removed, including misprediction penalties, penalties for redirecting sequential instruction fetch, and branch resource contention. Second, predication facilitates increased ILP and speedup by allowing separate control flow paths to be simultaneously executed. Figure 2(a) shows the performance when predication support is provided by the architecture and a capable compiler is employed to take advantage of it. The 6-issue processor simulated utilizes profile-based static branch prediction, a 4-cycle misprediction penalty, and a perfect memory system. Across all benchmarks predication yields an average performance gain of 34%.

**Predicate Utilization.** Although the performance of predicated execution is significant, it is at the cost of adding a predicate source operand on every instruction. In full predication model, all instructions have a predicate source operand, even those which are not conditionally executed. Figure 2(b) illustrates the percentage of static instructions with conditional predicates relative to the overall number of instructions. The percentage of conditional instructions averages around 40% of the total instructions, meaning that a large portion of instructions do not require a predicate operand. Since the percentage of unconditional instructions is significant, the unnecessary increase in instruction format size can dramatically impact embedded system designs.

**Predicated Instruction Cost.** Figure 3 shows the code size expansion attributed to the predicate operand for three distinct models on the same predicated benchmarks. First, *Zero Size* shows the code size for predication when the predicate representation has zero cost. Next, *Predicate Only* shows the effect when the instruction size growth of the predicate operand is attributed to only the conditional instructions. Finally, *Full Size* shows the size of the operand added to every static instruction as designed in an architecture supporting full predication. All of the predicated code sizes are compared to a base architecture without predication support. Note that compilation for predication alone has some effect on code size. The size of the predicate operand was evaluated assuming a 24-bit base instruction format and a 5-bit predicate operand field.

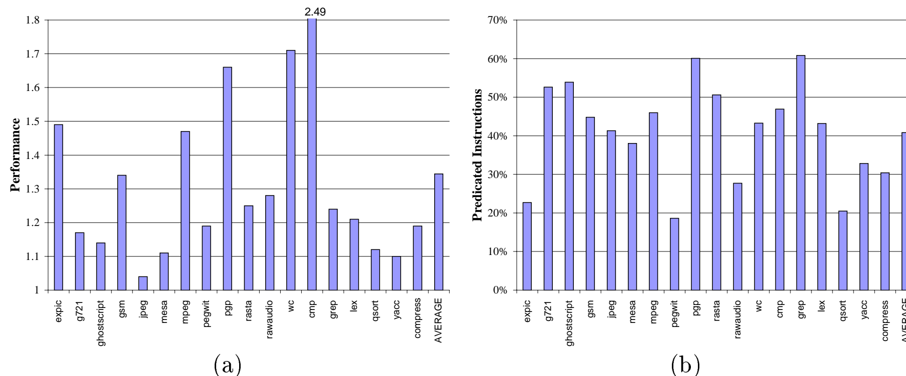


**Fig. 2.** Predicated execution performance (a) and utilization(b).

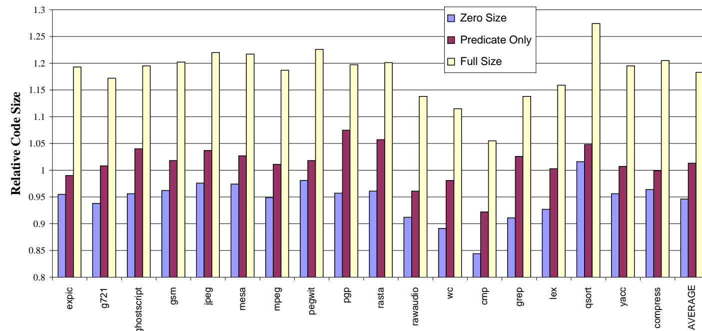


**Fig. 3.** Code expansion considering predication source operand.

Figure 3 indicates that predicated execution increases program code size by an average of 23%, and often as high as 30%. The results of the *Zero Size* model of code size evaluation indicate that for a large number of programs, predication effectively has fewer instructions and reduced code size. An interesting pattern is observed in Figure 3 for *Predicate Only* instructions. As a general rule, the code size for this model is significantly smaller than the *Full Size* code size, and averages near the base non-predicated code size. The difference between predicated and non-predicated results occur because predication has a fundamental ability to remove numerous control instructions and because compiler support of predicated execution can perform optimizations that allow the code to share instructions that are on different execution conditions. For example, in Figure 1(d) the instruction $D = A + X$ does not require a predicate operand since the compiler guarantees that it unconditionally executes in the block.

## 3 Prefix-Based Predication

There are several methods of adding predicated and non-predicated versions of instructions to an ISA design. This section details the addition of predication to a 24-bit instruction word for embedded processors. The proposed method is extendible to other instruction format constraints.

### 3.1 Architecture model

Prefix-based predication uses opcode prefixing to add sufficient instruction bits to indicate a predicate operand exists for instructions which the compiler has designated to conditionally execute. As illustrated in the previous section, a significant amount of code size can be saved when only the predicated instructions incur the predicate operand overhead. Figure 4 illustrates the base 24-bit instruction format that includes an operation code, a destination register index, and two source operands (potentially register indexes or immediate data).

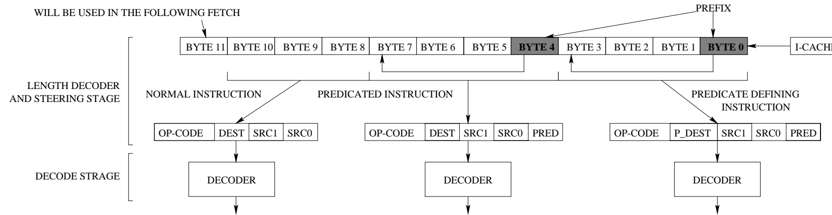


**Fig. 4.** Prefix-based predication decoding of normal and predicated instructions.

Figure 4 illustrates how a prefix opcode of the 24-bit instruction can designate that an additional 1-byte contains supplementary instruction information follows. The complete 32-bit instruction can then be decoded into a 26-bit instruction with a 6-bit operation code, a 5-bit predicate register index, a destination register index, and 2 source operands. The prefix opcode is then discarded. In this example architecture, the 5-bit predicate index can be used to access a 32-entry predicate register file. New predicate defining instructions for expressing predicate conditions are also added using the prefixng mechanism.

### 3.2 Microarchitecture support

The primary microarchitecture component affecting prefix-based predication is the instruction decode methodology. Most prefix architecture designs integrate an additional instruction decode stage in the original pipeline design. In this model, the first stage is used to determine instruction lengths (prefix detection) and steer the instructions to the second stage where the actual instruction decoding is performed. Figure 4 illustrates this process. The multiple pipelined decode method is successful for several reasons. First, the design places the focus on resources other than instruction memory. A second reason for using an additional decode stage is that the number of branch instructions executed in a predicated architecture is significantly reduced, resulting in the number of mispredictions also being reduced. This limits the negative effect of adding more pipeline stages before branch resolution has on the misprediction penalty. The branch prediction accuracy for predicated architectures is about 7% higher than branch prediction for traditional architectures.

## 4 Techniques for Reducing Predicated Code Size

The compilation techniques utilized in this paper to exploit predicated execution are based on an abstract structure called a *hyperblock* [7]. Several additions were made to the existing hyperblock framework of our experimental compiler. First, infrequently executed blocks with instruction merging opportunities that might normally be excluded from hyperblocks are included. This includes new methods of forming hyperblocks using basic blocks with zero or low execution frequency. New predicate optimization routines were also developed.

The new optimization routines extend the techniques of predicate promotion and predicate merging. Predicate promotion refers to speculation performed by changing an instruction's predicate to a predicate whose expression subsumes that of the original predicate [7]. Promotion may result in the instruction being unconditionally executed, reducing the number of predicated instructions. Predicate merging allows identical instructions on intersecting predicate conditions to be combined. Merging thereby removes one instruction copy, and promotes the remaining instruction. These optimizations will be detailed in a future work, and are not presented here due to space limitations.

### 4.1 Predication Code Size and Execution Characteristics

The compiler's ability to affect program code size and the percentage of predicated instructions for several benchmarks is now examined. The selected benchmarks are the MediaBench suite [5] and UNIX utilities. Figure 5 indicates that predication reduces the total number of instructions for traditionally optimized code by 6.3%. A significant portion of the instructions eliminated were control instructions, which were reduced by 13%, where control instructions include predicate defining instructions and any traditional branch instructions. Other characteristics include a 7% reduction in the number of dynamically executed instructions in general code, and a 31% reduction in the number of dynamically executed instructions in code with superscalar optimization.

Table 1 summarizes the amount of predicate optimization that the compiler is able to perform on the hyperblocks. For the instruction merging category, the percentage of static predicated instructions averages 8% that can be merged. The additional code reduction attributed to merging is shown in the next column. The percentage of predicated instructions that are promoted to unconditionally executed instructions is shown in the next column. These numbers indicate that an average 28% of the originally predicated instructions may be promoted. The final two columns include the percentage of static predicated instructions

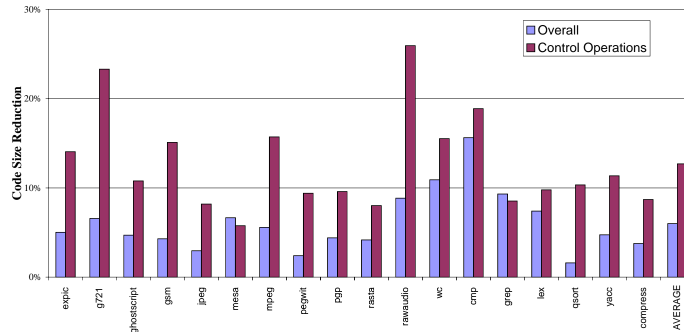


**Fig. 5.** Code reductions due to predicated execution.

| Benchmark | Merging % | Code Merging Reduction % | Promotion % | Hyperblock Static Pred % | Pred-Optimization Static Pred % |
|---|---|---|---|---|---|
| expic | 6.03 | 1.45 | 37.59 | 22.68 | 14.92 |
| g721 | 1.60 | 0.85 | 43.31 | 52.64 | 29.77 |
| ghostscript | 0.26 | 1.01 | 32.68 | 41.31 | 24.79 |
| gsm | 3.21 | 1.80 | 51.44 | 44.78 | 23.28 |
| jpeg | 29.97 | 1.96 | 39.38 | 53.88 | 34.78 |
| mesa | 8.62 | 3.55 | 37.49 | 37.96 | 22.07 |
| mpeg | 5.05 | 2.40 | 34.52 | 46.03 | 26.13 |
| pegwit | 3.72 | 0.75 | 15.08 | 18.60 | 14.95 |
| pgp | 2.48 | 1.52 | 14.12 | 60.12 | 49.32 |
| rasta | 3.38 | 1.75 | 17.48 | 50.60 | 39.94 |
| rawaudio | 2.17 | 0.61 | 26.09 | 27.71 | 21.21 |
| wc | 16.92 | 7.91 | 10.77 | 43.33 | 40.29 |
| cmp | 22.12 | 11.57 | 16.81 | 46.89 | 37.04 |
| grep | 10.52 | 6.89 | 14.43 | 60.85 | 50.74 |
| lex | 11.87 | 5.44 | 14.97 | 43.15 | 32.75 |
| qsort | 8.00 | 1.61 | 48.00 | 20.49 | 11.90 |
| yacc | 7.30 | 2.48 | 26.26 | 32.83 | 21.11 |
| compress | 5.85 | 1.82 | 30.70 | 30.37 | 14.60 |
| average | 8.28 | 3.08 | 28.40 | 40.79 | 28.31 |

**Table 1.** Instruction merging and predicate promotion characteristics.

relative to total program instructions for the original and predicate-optimized hyperblocks. The most important result of Table 1 is that only 28% of the static instructions remain predicated after predicate optimization.

## 5 Experimental Evaluation

### 5.1 Methodology

The IMPACT compiler and emulation-driven simulator were enhanced to support the proposed architecture framework. The base architecture modeled uses a 5 stage pipeline that can issue in-order 6 operations per cycle (up to the limit of the available functional units: four integer ALU's, two memory ports, two floating point ALU's, and one branch unit). The instruction latencies used match the HP PA-7100 microprocessor (integer operations have 1-cycle latency, and load operations have 2-cycle latency). The processor contains 32 integer and 32 floating point registers. To support prefix-based predication, 32 predicate registers and an additional decoding stage were modeled. The memory system simulated was either perfect or used a 2K, 4K, or 8K sized direct-mapped instruction caches and a 8K direct mapped, blocking data cache; both with 64-byte blocks and a miss penalty of 12 cycles. A static branch prediction strategy was employed.

### 5.2 Results and Analysis

Figure 6 shows the results of varying the instruction cache size for the non-predicated and prefix-based predicated architectures. Substantial performance improvement is established at small cache sizes; however, for larger increases in instruction cache size, the relative performance improvements of the base architecture are larger, and the relative performance saturates. This indicates that

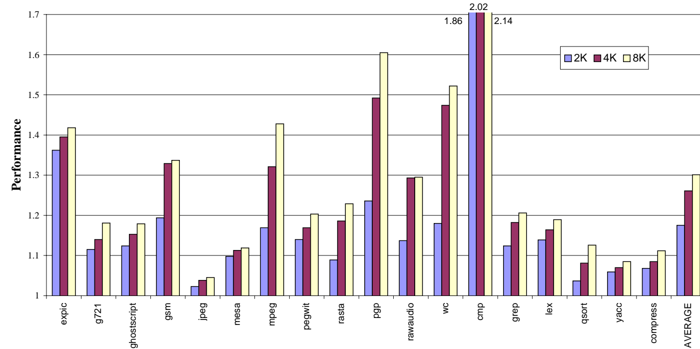


**Fig. 6.** Performance of varying instruction cache size for prefix-based predicated architecture relative to non-predicated architecture.

the base model is more dependent on instruction cache resources than the prefix-based predicated architecture. The results of cache simulations show that prefix-based predication has an average 7% higher hit rate for 2K instruction caches and 2.5% for 8K caches compared to the non-predicated model. Experiments also indicate that prefix-based predication has an average 10% higher speedup over traditional predicated architectures for small instruction cache models.

The relative performance of superscalar (superblock formation, loop unrolling) optimization for prefix-based predicated and non-predicated architectures is an average 63% better than general levels of optimization for the simulation of a perfect memory system. For superscalar optimization, the average speedup of the predicated architecture is only 12% more than the non-predicated architecture. The performance of the superscalar optimization indicates that the performance gains of predicated execution do not greatly exceed the non-predicated version. However, the corresponding code size of the predicated code

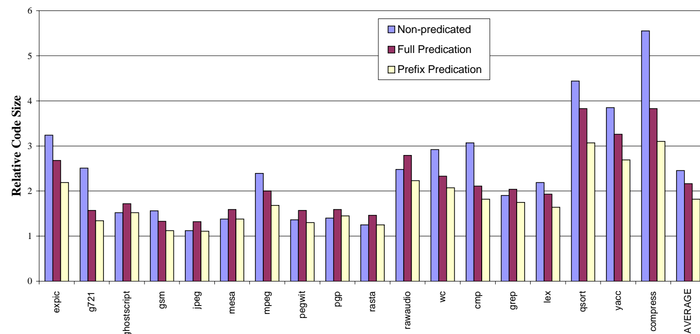


**Fig. 7.** Code expansion of superscalar relative to traditional optimization.

for high performance code is significantly reduced. Figure 7 shows the code expansion of the superscalar optimization for the non-predicated, full-predicated, and prefix-based predicated architectures. Clearly the 12% performance improvement is substantial since the improvement requires a significantly smaller code size. The full predicated architecture has an average 11% smaller code size and the prefix-based predicated architecture has an average 25% smaller size.

## 6 Conclusions

The prefix-based predicated execution architecture framework proposed has the potential to significantly enhance the effectiveness of introducing predicated execution into embedded microprocessors. For regions of non-predicated code, the prefix-based method offers better code density characteristics than traditional models of predication support. For predicated regions, the prefix-based method offers performance improvement over an architecture without predication support. We illustrate that an optimizing compiler can enhance the prefix-based predication model by performing aggressive instruction merging and predicate promotion to reduce the number of predicated instructions by 30%. Overall, prefix-based predication achieves 12% performance improvement for code created with superscalar optimization and reduces code size by 25%.

## References


1. J. Davidson and R. Vaughan. The effect of instruction set complexity on program size and memory performance. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 60–64, October 1987.
2. R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits*, 31:1277–1284, 1996.
3. W. W. Hwu et al. The Superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1):229–248, January 1993.
4. V. Kathail et al. HPL PlayDoh architecture specification: Version 1.0. Technical Report HPL-93-80, Hewlett-Packard Laboratories, Palo Alto, CA, February 1994.
5. C. Lee and W. Mangione-Smith. Mediabench In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 330–335, December 1997.
6. S. A. Mahlke et al. A comparison of full and partial predicated execution support for ILP processors. In *Proceedings of the 22th International Symposium on Computer Architecture*, pages 138–150, June 1995.
7. S. A. Mahlke et al. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 45–54, December 1992.
8. MicroDesign Resources. *Embedded Processor Forum*, San Jose, CA, October 1998.
9. J. C. Park and M. S. Schlansker. On predicated execution. Technical Report HPL-91-58, Hewlett Packard Laboratories, Palo Alto, CA, May 1991.
10. D. N. Pnevmatikatos and G. S. Sohi. Guarded execution and branch prediction in dynamic ILP processors. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 120–129, April 1994.