# Lightweight Predication Support for Out of Order Processors

Mark Stephenson, Lixin Zhang, and Ram Rangan
IBM Austin Research Lab
11501 Burnet Road, Austin TX 78759, USA
{mstephen,zhangl,rrangan}@us.ibm.com

## Abstract

*The benefits of Out of Order (OOO) processing are well known, as is the effectiveness of predicated execution for unpredictable control flow. However, as previous research has demonstrated, these techniques are at odds with one another. One common approach to reconciling their differences is to simplify the form of predication supported by the architecture. For instance, the only form of predication supported by modern OOO processors is a simple conditional move. We argue that it is the simplicity of conditional move that has allowed its widespread adoption, but we also show that this simplicity compromises its effectiveness as a compilation target. In this paper, we introduce a generalized form of hammock predication — called predicated mutually exclusive groups — that requires few modifications to an existing processor pipeline, yet presents the compiler with abundant predication opportunities. In comparison to non-predicated code running on an aggressively clocked baseline system, our technique achieves an 8% speedup averaged across three important benchmark suites.*

## 1 Introduction

Single-thread performance is extremely important in many computing communities. In accordance with Amdahl's law, poor single-thread performance can even hinder the performance of parallelized applications that contain modest sequential regions. Out of Order (OOO) execution is one popular method to extract higher levels of instruction-level parallelism and thereby increase single-thread performance. While OOO execution provides performance benefits, it only serves to exacerbate performance inefficiencies related to branch mispredictions; and unfortunately, many branches elude even the most sophisticated prediction mechanisms.

Architects have proposed approaches that effectively remove hard-to-predict branches. Common to these techniques is the ability to process multiple control flow possibilities in advance and later, upon resolution of branch conditions, selectively discard instructions from the wrong paths. Dual-path execution [9], poly-path execution [14], dynamic predication [13], and static predication are among the most popular of these mechanisms. The first three mechanisms assume little to no compiler support. Consequently, the hardware is responsible for fetching multiple control flow paths, enforcing correct dependence flow between instructions, and discarding wrong path instructions. Though recent research has shown these approaches can substantially increase performance, the architectural modifications involved may preclude their commercial adoption.

In contrast, many OOO processors already feature *conditional move*, a simple form of static predication that moves the contents of one register to another register subject to the value of a guarding condition (*e.g.*, [11], [10]). For instance, the Alpha 21264 converts a conditional move into an internal *ternary* operator, `rd = cond ? rs : rd`, that depending on the value of `cond`, either copies the value in register `rd` to itself, or assigns it the value in register `rs` [11]. Because of the restrictive nature of this form of predication, the implementation of conditional move requires only slight micro-architectural modifications.

Similarly, this paper proposes a method of static predication for OOO processors that restricts the *form* of predicatable regions in order to simplify the hardware support beyond that required by prior art; but unlike conditional move, our approach still allows for predication of memory operations, arbitrary integer and floating point arithmetic instructions, and several other (potentially excepting) instructions.

Static predication is an approach that relies on compiler technology to transform control flow dependencies into data dependencies [2, 16, 18]. In doing so, static predication can merge several disparate paths of control flow into a single instruction stream. While predication obviates the need for hardware to simultaneously fetch more than one control flow path, it still relies on hardware to discard wrong path instructions, and more importantly, to enforce correct dependence flow. The latter requirement is not straightforward to fulfill in OOO processors, and simplifying this problem is the focus of our approach.

OOO processors possess a mechanism to identify the producer of a register operand (*e.g.*, the operation that writes to a given register), and a means to "wakeup" any instructions waiting to consume that operand. The register re-

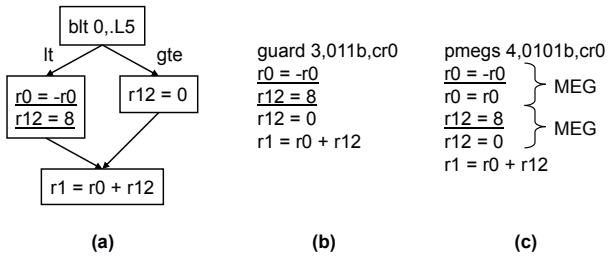


**Figure 1. Predication example.**

namer, which associates physical registers with unique producers, allows the processor to identify instructions that are ready to issue. However, predication introduces conditional writers, and hence obfuscates the "wakeup" process and bypass logic in OOO processors.

Figure 1 provides an example that highlights the complication. Part (a) shows a simple hammock from the multimedia benchmark `adpcm`. Because this paper is primarily concerned with lightweight predication support, we use an instruction set architecture (ISA) extension similar to that introduced in [19]. The `guard` instruction in part (b) predicates the three instructions that follow it according to the bitmask "011". The underlined operations correspond to instructions that have a '1' in the guard, and thus the results of these instructions are only committed if the corresponding condition register bit (`cr0`) is 1. Likewise, the third instruction guarded by the `guard` instruction only commits its results when the condition register bit is 0.[1]

Notice that the producers of the source operands for the addition instruction (`r1 = r0 + r12`) in part (b) are ambiguous. Until the condition register bit has been resolved, the processor does not know which instructions will produce the source operands. While it is possible (and fruitful) to dynamically keep track of dependence flow information in the presence of predicated code regions, implementing such approaches requires modifying much of a processor's pipeline.

Our approach simplifies the hardware's dependence tracking responsibilities by ensuring that predicated regions satisfy three major invariants: the compiler guarantees that 1) any predicated region of code is composed of two *mutually exclusive* paths of control; 2) that there is an *ordered*, one-to-one correspondence between the register writes on both paths of control flow in the predicated region; and 3) that instructions from the two paths are perfectly interleaved. While this may seem restrictive, we show that it presents over five times as many predication opportunities than does conditional move support.

Figure 1(c) provides an example of how the compiler would generate code using our approach. The first two instructions comprise a *mutually exclusive group* (MEG). They both write to the same ISA register, and we are guaranteed that only one of them will commit its result. Notice that the compiler has to insert a "nop move" instruction to satisfy the aforementioned constraints of a MEG. Likewise, the last two instructions in the guarded region comprise a MEG, but here the compiler is able to "overlap" useful instructions from the two paths of control flow. In this paper, we refer to such regions as *predicated mutually exclusive groups* (PMEGs) and we use the mnemonic `pmegs` to disambiguate such regions from those presented in [19]. As we describe later in this paper, the processor can treat MEGs as single units and trivially keep track of dependence flow information.

The novel contributions of this paper are as follows:

- We show how to minimally augment an OOO processor such that it can execute PMEGs.
- We present a compiler algorithm that, assuming a fixed instruction schedule, optimally "interleaves" predicated regions of code. As we empirically demonstrate, the compiler finds a substantial amount of overlap when interleaving disparate control flow paths.
- We demonstrate the value of `guard` and `pmegs` in the context of an OOO processor by evaluating their performance gains on a wide variety of benchmarks. Our technique outperforms non-predicated baseline code by 8%, and nearly matches the performance of an unrealistically aggressive comparison model for hammock predication.

PMEGs execution is not limited to two paths, nor to `guard`-style predication. While we chose `guard` as a baseline because of its simplicity, the concepts we present in this paper could readily be extended to support more complex control flow.

## 2 Related Work

This section describes previous attempts to provide multi-path execution support for OOO processors. The simplest form of predication support for OOO processors is the so-called *conditional move* instruction. Even though this support is highly constrained — it allows only a single register move operation to be conditionally executed — it has been shown to be useful for certain classes of applications [17, 21, 3]. In part because of the simplicity of this approach, such support has been in commercial processors for decades. However, compilers often cannot determine the safety of if-converting with conditional move. For instance, to if-convert the following C code using conditional

[1] For the remainder of the paper, we use *predication* and *guarding* interchangeably to indicate the concept of conditionally executing a region of code. Likewise, we will use either *guarded* or *predicated* to indicate that a set of instructions conditionally execute, and we say that a region is either *predicatable* or *guardable* if the compiler can predicate it.

move, the compiler must ensure that neither of the memory loads will throw an exception:

```
y = (a < b) ?  A[i] :  B[i].
```

Klauser et al. proposed a dynamic predication scheme that focuses on the types of regions with which this paper is concerned— simple hammocks [13]. In [13], the register renamer is augmented to hold three separate mappings for each producer, and the renamer also dynamically injects conditional move instructions to merge in the results at join points. They report an average speedup of 5.5% and up to 13% in the benchmarks they studied. Their work shows the potential that predicating simple hammocks can bring. We compare against a similar approach in this paper.

More recently Wang et al. explored a hypothetical Itanium™ design in which instructions can issue out-of-order [26]. To deal with register renaming conflicts their design dynamically injects "select" $\mu$-ops to merge data from different paths. They found the OOO design to be 16% better than the corresponding in-order design, 7% of which they attributed to efficient predication support. However, this design point complicates rename and issue logic, requires the addition of a select function unit, and necessitates the introduction of a select $\mu$-op (essentially a conditional move) for every predicated writer. In the same work, the authors also introduce the notion of a "predicate slip", another design point in which predicated instructions are allowed to issue, execute, and write back results before the guarding predicate has been resolved.

Darsch and Seznec introduce a Translation Register Buffer (TRB), which adds an extra level of indirection to register accesses, to support out-of-order issue for IA64 codes. The TRB maps an IA64 ISA register to either a physical register *or* a logical IA64 register [7]. When multiple predicated writers are in flight, this technique involves traversing the TRB entry chain backward to determine the last good writer. Even though the TRB increases performance by 10%, its high algorithmic complexity may impede a practical implementation.

The Diverge Merge Processor (DMP) of Kim et al. uses compiler support to demarcate, via ISA extensions, the basic blocks in which control flow splits, as well as the corresponding confluence points [12]. For low-confidence branches that have been tagged as "diverge" points, the processor forks the computation and register rename tables; at merge points, the main thread of execution is reconciled with the correct path of control. While they demonstrate impressive speedups ($\sim$20%) over a deeply-pipelined, aggressive out-of-order processor, the architectural complexity of the DMP is significant.

Predicate prediction [5] predicts the value of a predicate in the dispatch logic and speculatively executes one path of predicated instructions. An efficient implementation of predicate prediction relies on replay support.

TRIPS presents an interesting model for merging predication with OOO processing [23]. Though the TRIPS architecture is vastly different from the types of architectures our research targets, the TRIPS ISA similarly constrains its predicated regions. To facilitate the identification of block completion, TRIPS requires that all executions of a predicated block produce the same set of register writes. For an entirely different reason we enforce a similar requirement on MEGs (namely, to reduce the complexity of the register renamer and issue logic). We note that the requirement we place on MEGs is more stringent because the compiler must consider the order of predicated instructions. While a simple dataflow analysis can identify the set of writers on a given path, optimally interleaving the writers requires a more sophisticated approach.

Finally, Sprangle and Patt describe a creative basic-block-atomic execution scheme in which the compiler explicitly encodes data dependencies [25]. This point in the design space requires several extra bits of encoding in the ISA and places extra burden on the compiler. Similar to our approach, the authors mention that such a scheme could allow for efficient predication support by assigning the same tag to certain mutually exclusive instructions. However, they do not discuss how this could be done in software, nor do they demonstrate the efficacy of their approach.

The survey of related work shows the promise of predication in the context of OOO processing. Much of the related work we discuss here supports full, arbitrarily complex predicated regions [7, 26, 23]. Our compiler-directed technique focuses on hammocks and requires only minimal changes to the renamer and the issue logic to efficiently support predication. Much as the simplicity of the conditional move instruction allows for simple integration in modern OOO processors, the approach we describe in this paper minimizes hardware modifications, while dramatically expanding the compiler's ability to remove branches.

# 3 Our Approach

Before describing microarchitectural mechanisms, we discuss the high-level idea of our approach, and present an optimal compiler algorithm for generating minimally sized predicated regions. The ISA extension discussed in [19] is the foundation for our predication support, and as such, we begin by describing the programming model for this "baseline" approach.

## 3.1 Baseline Predication Support : Guard

In [19], Pnevmatikatos and Sohi show that one can add lightweight predication support to an existing ISA with the addition of a single instruction. In this paper we augment the PowerPC ISA with a similar instruction, which as in [19], we call `guard`. The `guard` instruction is a predicate-defining instruction that begins a predicated block

of instructions; it specifies the length of the predicated region, a bitmask that discriminates between two possible paths of control flow, and a condition code bit that dictates which instructions to execute.

To recall a concrete example, the `guard` instruction in Figure 1(b) guards the subsequent three instructions. The binary bitmask, "011", specifies that the first two instructions (corresponding to the low order two bits) belong to the *true* path, while the last instruction belongs to the *false* path. Thus, if the condition code bit `cr0` is true, the results of the first two guarded instructions (which are underlined in the figure) will commit, while the last guarded instruction's (`r12 = 0`) result will be discarded (and any exceptions that it might cause will be ignored). Akin to [19], the bitmask, the condition code bit, and the length of the region become part of the architectural state that must be saved and restored on interrupts, exceptions, and context switches.

Guarded regions can contain all instruction types except for `guard` instructions (*i.e.*, for simplicity's sake we only consider simple control flow hammocks). To further simplify the required hardware and software support, we enforce the following constraints on guarded regions:

- A guarded region contains *exactly* two mutually exclusive paths of control flow.
- A guarded region has between 1 and 15 instructions.
- A guarded region *can* contain jumps and branches, but only if the corresponding jumps and branches *end* the region.

Even under such constraints guarding opportunities abound in typical code. It is very important to note that, unlike conditional move, `guard` can execute arbitrary integer and floating point arithmetic instructions, load and store instructions, and instructions that operate on condition code registers. It is common to find such instructions in typical hammocks, and thus, in terms of scope, `guard` is superior to conditional move.

### 3.2 Predicated Mutually Exclusive Groups

Unfortunately, supporting the `guard` instruction in an OOO processor can require several non-trivial design modifications. Our approach, which we take specifically to simplify dependence checking and the associated wakeup logic, places the following two additional constraints on predicated regions:

- The sequence of register writes is the same on both paths.
- Instructions from both paths are perfectly interleaved.

We call guardable regions that satisfy these constraints — such as the region in Figure 1(c) — *predicated mutually exclusive groups* (PMEGs). PMEGs are therefore comprised of at least one mutually exclusive group (MEG), where each MEG contains a pair of instructions from mutually exclusive paths. Furthermore, both instructions in a MEG necessarily write to the same ISA register. Because the architecture knows that *exactly* one instruction from a MEG will commit its results, the register renamer can provide the same register mapping to both instructions. As Section 4 explains, this property enables several hardware simplifications relative to prior art.

Code generation for regions predicated with `guard` is fairly straightforward. For a given region, the compiler removes the associated branch instruction(s), and then merges the *then* block with the *else* block (if present). Finally, the compiler generates the `guard` instruction — with an appropriate bitmask — to guard the newly-formed block.

PMEGs, however, require more care. The compiler must ensure that every region meets the constraints for PMEG execution, and thus the compiler must exclude some regions that it could predicate with `guard`. For instance, some ISA instructions write to multiple registers. The compiler can often split these instructions into several single-writer instructions, but in other cases, splitting is not feasible. Another potential issue is that a "nop move" instruction — which we define as an ISA instruction that moves a register to itself — may not exist for some special registers. Because the PowerPC ISA features a rich instruction set, this was not an issue for the experiments we perform in this paper. As we show in Section 6, around 90% of the regions for which `guard` is applicable, `pmegs` is also applicable.

For *if-then* regions, the constraints are trivially met by introducing a nop move instruction for every instruction in the *then* block, and interleaving the instructions appropriately. *If-then-else* regions however, can often be overlapped. Given a fixed schedule — an appropriate constraint since the instruction scheduler runs before the guarding pass — the compiler can optimally overlap instructions from the *then* and *else* blocks (*i.e.*, minimize the size of the resultant PMEGs region) by leveraging solutions to the *longest common subsequence* (LCS) problem. The goal of the LCS problem is to find ordered correspondences between two sequences. Formally stated, given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, and a sequence $Y = \langle y_1, y_2, \ldots, y_n \rangle$, the LCS is a *strictly increasing* sequence of indices $I = \langle i_1, i_2, \ldots, i_k \rangle$ of $X$, and indices $J = \langle j_1, j_2, \ldots, j_k \rangle$ of $Y$ such that $\forall l = 1..k$, $x_{i_l} = y_{j_l}$. Here the length of $I$, $||I||$, is maximal and $||I|| = ||J||$ [6].

Figure 2 depicts an *if-then-else* region that demonstrates the applicability of the LCS problem to forming PMEGs. In part (a), the compiler creates two sequences for the *then* and *else* paths respectively: $\langle r5, r3, r9, mem \rangle$, and $\langle r5, r6, r3, r9 \rangle$. Solving the LCS problem yields the associated sequences of indices, $\langle 1, 2, 3 \rangle$, and $\langle 1, 3, 4 \rangle$, for the *then* and *else* paths respectively. These sequences of

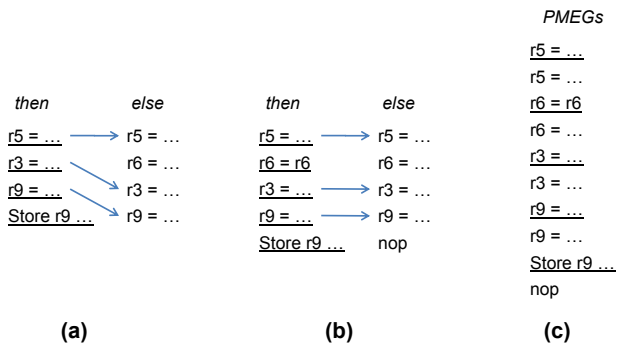


**Figure 2. Optimally overlapping instructions.**

indices correspond to the longest common subsequence $\langle r5, r3, r9 \rangle$. As we show in part (b), before merging the *then* block and the *else* block to form a `pmeg` region, the compiler inserts nop moves where needed to enforce the property that both paths have the same sequence of register writes. If there is a corresponding register write in both blocks, no nop move is needed. However, if one block contains a register write that the other does not, the compiler adds a nop move to the deficient block to balance the paths. As shown in the figure, the compiler also tries to pair instructions that do not write to a register (*e.g.*, a store instruction); if the compiler cannot find a match, it will insert a *true* `nop` instruction. Finally, as in part (c), the compiler interleaves the instructions from the two blocks to form a single predicated block.

There exists an efficient (O($n + m$), where $n$ and $m$ are the lengths of the two sequences respectively) dynamic programming based algorithm that optimally solves this problem; this well-known result is detailed in [6]. The PMEGs approach allows the compiler to create almost as many predicated regions as `guard`, and far more regions than conditional move.

### 3.2.1 Compiler Implementation

For the experiments we present in this paper we use GCC version 4.1.2 [8], which we modified by adding a *guarding* pass, and by bolstering its support for PowerPC's conditional move instruction, `isel`. The `isel` (integer select) instruction is a slightly more flexible variant of conditional move [17]. As in [17], our compiler can generate certain conditional memory stores and loads using `isel`. In addition, we modified binutils 2.16.1 by adding the definition of `guard` and `pmegs`. Our compiler pass runs extremely late in the compiler's toolchain, just slightly before assembly code is generated, but before the compiler stops maintaining the control flow graph. At this point in the toolchain, our pass will not interfere with downstream compiler optimizations, and we also have the benefit of knowing exactly how many instructions are within a guarded region (and because guarded regions can contain at most 15 instructions, this is important).

The guarding pass can generate code for one of two different models: the baseline `guard` model, and the simpler (from a microarchitectural perspective) `pmegs` model. The guarding pass first identifies simple hammocks (which correspond to *if-then* and *if-then-else* regions) in the control flow graph, and then determines the subset of such hammocks that the compiler can legally predicate. The guarding pass assumes that it is profitable to predicate a `guard` or a `pmegs` region that has fewer than seven instructions and that also has a maximum dependence height of three or less.

Intuitively, this heuristic makes sense: shorter regions are likely to contain fewer dynamically useless instructions. We empirically determined that predicating regions longer than seven instructions is often detrimental to performance. Likewise, we noticed that highly sequential regions (corresponding to regions with large dependence heights) are far less likely to benefit from predication. Predicated instructions cannot commit results until their associated predicate values resolve. Instructions that depend on a predicated sequential computation are delayed until the predicate is resolved *and* the entire dependence chain is committed. While more sophisticated cost functions (including functions that consider the predictability of the associated branches) will provide added benefits, we show that this straightforward heuristic leads to significant speedups.

## 4 Hardware Support

We now describe the hardware requirements for supporting `guard`-style predication in an OOO processor. We first lay the groundwork for describing our PMEGs approach by outlining baseline hammock predication implementations.

In a typical OOO pipeline, the *front end* — which includes the fetch unit, decode unit, and rename unit — processes instructions in program order. The *back end*, on the other hand, supports OOO processing: the issue unit sends "ready" instructions, irrespective of program order, to an appropriate execution unit. The reorder buffer then reserializes computation so that the commit unit can commit instructions in program order.

The literature considers two main approaches for supporting predication in an OOO processor. The first approach stalls predicated instructions in the issue unit until their associated predicate conditions are resolved [25, 26]. Stalling consumes issue queue slots, delays the computation of the predicated region, and stalls any subsequent instructions that have source operands written in said region. However, it avoids wrong-path effects and simplifies the commit logic by discarding useless instructions in the issue stage.

A more aggressive approach allows predicated instructions to issue, execute, and write back results (to renamed

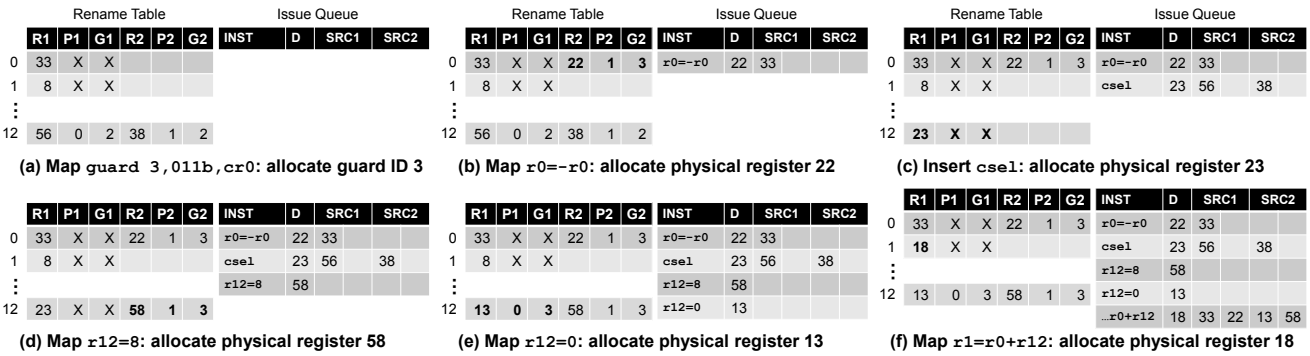


**Figure 3. Renaming the region in Figure 1(b). See text for details.**

registers) before the associated predicate conditions are resolved [13, 26]. The commit logic discards predicated-off instructions and commits predicated-on instructions.[2] We refer to this approach as *stall-at-commit*. Common to both *stall-at-issue* and *stall-at-commit* is the need to maintain sufficient bookkeeping for handling multiple outstanding `guard` instructions, recovering from pipeline flushes and other exceptions, and supporting context switches.

Before discussing the hardware mechanisms required for PMEGs execution, we first outline the support required for *stall-at-issue* and *stall-at-commit* in the context of `guard`-style predication. We conclude this section with a qualitative discussion of hardware complexity.

## 4.1 Guard with Stall-at-Issue

Here we describe our specific, idealized implementation of the *stall-at-issue* mechanism described in [25] and [26]. *Stall-at-issue* allows predicated instructions to pass through the rename unit into the issue unit before their associated predicate conditions resolve. Predicated instructions are held in the issue unit until the processor determines the value of the guarding predicates, at which point the processor nullifies the predicated-off instructions and issues the predicated-on instructions to appropriate execution units. While this approach leads to excellent performance improvements, it requires several changes to the rename and issue logic.

The rename unit in a superscalar pipeline manages a rename table that associates physical registers with logical registers. Typically, each issue queue entry associates an (internal) instruction with its physical source and destination operands; the issue logic consults the issue queue to determine when an instruction's operands are available.

To support `guard`-style predication, we use rename table extensions similar to those presented in [13, 26]. Figure 3 shows the step-by-step state of the rename table for the example from Figure 1(b). The rename table now keeps track of up to two physical registers per entry (R1, R2). For each register, the table saves its physical register number (R1, R2), a flag indicating whether or not it is the destination of a predicated instruction (P1, P2), and a tag that uniquely identifies the guarding `guard` instruction (G1, G2). The figure uses an 'X' for the predication flag and guard identifier of unconditional writers.

The augmented rename table allows the processor to associate multiple physical registers with a given logical register as in [13, 26]. In addition, we must extend each issue queue entry so that it can track up to two predicated logical registers for each instruction operand. Note that in [26], rather than augment the issue queue, the authors inject so-called $\mu$-ops into the pipeline that select between two possible physical registers. Thus, our specific implementation of *stall-at-issue* adds complexity, yet it provides an upper bound for *stall-at-issue* performance in the context of `guard`-style predication.

Each predicated register in the rename table contains a tag that specifies which `guard` instruction predicates the register. The unique identifiers are assigned to the `guard` instructions in the rename stage and recycled on commits and pipeline flushes. The renamer stalls when it receives a `guard` instruction and all identifiers are in use.

As Figure 3 illustrates, we use a policy similar to [26] for updating the rename table. As in Figure 3(b), for a given instruction, if the corresponding rename table entry only has one occupied slot — which must contain a non-predicated register — it updates the other slot.

However, it is possible for both entries to be occupied by physical registers that are the destinations of unresolved predicated instructions controlled by a different `guard` instruction. In this case, as in [26], the renamer inserts a conditional move operation (`csel`) to make space in the rename table for new guarded destinations. The renamer will allocate a new physical register for the `csel` instruction, and update the corresponding rename table entry with

[2] We call predicated instructions with a dynamically *true* guarding predicate (*i.e.*, the instruction's results can safely commit), predicated-on. Likewise, we refer to predicated instructions with dynamically *false* guards, as predicated-off instructions.

the allocation. In Figure 3(c), the renamer identifies such a situation when trying to allocate a physical register for the instruction `r12=8`. Because both register mappings are occupied for `r12`, the renamer injects a `csel` instruction. The rename unit can then proceed to handle the stalled predicated instruction (Figure 3(d)). It is important to note that in our implementation, we ideally assume that the `csel` instructions instantaneously execute; this simulation point is useful because it provides us with an upper bound on attainable performance.

Again in Figure 3(e), both table entries for logical register 12 are occupied. However, in this case, the rename unit simply replaces the non-predicated register entry with the newly allocated physical register 13. In this case the augmented renamer knows — based on the shared guard identifier — that these destination registers are mutually exclusive and exactly one of them is guaranteed to commit.

After the rename stage, predicated instructions enter the issue unit. As soon as a predicated region's predicate condition resolves, associated predicated-off instructions will be removed from the issue queue and marked as "complete" in the reorder buffer; the predicated-on instructions will become ready-to-issue candidates as soon as their source operands are available. In addition, the processor must update the rename table: physical registers allocated to the predicated-on instructions are marked as non-predicated, and entries associated with the predicated-off instructions are removed. All memory instructions are inserted into the load-store queue at the same time that they are moved into the issue unit, which enables the load-store queue to keep requests in program order, thereby facilitating address conflict detection. Predicated memory operations are removed from the load-store queue if they are determined to be predicated-off. If a predicated instruction reaches the rename unit after its guarding predicate resolves, the rename unit will discard this instruction if it is predicated-off or handle it as a normal non-predicated instruction if it is predicated-on.

Non-predicated instructions that need to read registers written by unresolved predicated instructions will take both registers in the rename table as potential source mappings for its source operands (see Figure 3(f)). Traditionally, the issue logic watches for one physical register for each source operand. For the *stall-at-issue* support, the issue logic is augmented to snoop for two physical registers for each source operand. If either one of the two physical registers becomes available, the issue logic will mark the source operand as *ready*.

Even though *stall-at-issue* adds complexity to the rename and issue units, we assume an optimistic implementation that contains the same number of pipeline stages and the same clock cycle time as the baseline and the PMEGs model. In Section 6 we show that the PMEGs approach compares favorably to the upper-bound *stall-at-issue* support described above.

### 4.2 Guard with Stall-at-Commit

The *stall-at-commit* approach is a more aggressive extension of *stall-at-issue*. It allows guarded instructions to issue, execute and write back results even before their associated predicate conditions are known. It therefore makes more efficient use of the window between when a predicated instruction is renamed, and when its corresponding `guard` instruction is executed. If the window is sufficiently large, a predicated instruction can execute before its associated `guard` instruction.

This approach is investigated in [13] and [26]. The premise of the approach is that, on predicate-aware architectures, predicated-off instructions can execute without modifying machine state. *Stall-at-commit* leverages this property to speculatively execute predicated instructions from disparate paths of control flow, while making certain that it only commits the results of the predicated-on instructions.

While [26] shows that this approach leads to marginal performance improvements over *stall-at-issue* ($\sim$1%), there are some cases where it degrades performance. Notably, *stall-at-commit* can suffer from *wrong-path* effects, such as stalling on a long-latency predicated-off load miss. In addition, if the predicate condition is resolved early, because *stall-at-issue* summarily discards predicated-off instructions, *stall-at-issue* will execute more efficiently than *stall-at-commit*. In addition to extending the rename and issue units, this approach requires extra logic to handle predicated instructions in the commit unit and the load-store unit.

### 4.3 PMEGs with Stall-at-Issue

We are now in the position to present the hardware implementation for our PMEGs approach. Our approach minimizes the number of architectural modifications required for supporting predication. To briefly recap, the compiler ensures that predicated regions always consist of perfectly "interleaved" instructions from two mutually exclusive paths of control flow. From an architectural perspective, the key advantage of PMEGs is that it requires no changes to the rename table and only minor modifications to the rename unit and issue logic found in typical (predicate-unaware) processors.

Our specific implementation requires an instruction from the *true* path to precede the *false*-path instruction in each PMEG. For each PMEG, if the predicated-*true* instruction writes a register, the rename logic allocates a physical register for this instruction (Figure 4(a) and (c)). However, the renamer does not update the rename table until it receives the ensuing predicated-*false* instruction, which reuses the physical register allocated for the preceding predicated-*true* instruction (Figure 4(b) and (d)).

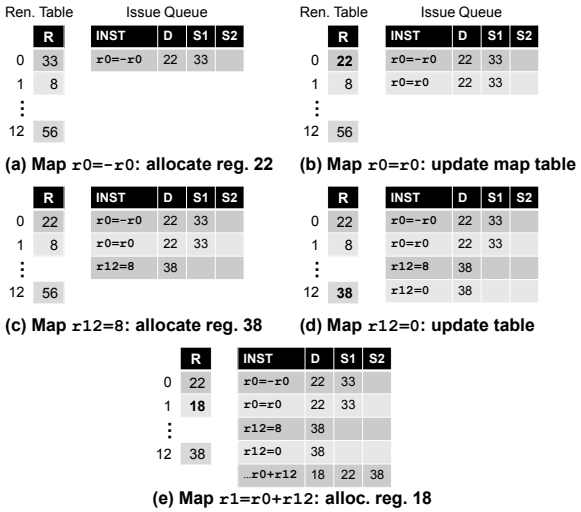


**Figure 4. Renaming the region in Figure 1(c).**

The delayed rename table update is necessary because the predicated-*false* instruction needs the mapping information before the predicated-*true* instruction is renamed. For example, as we show in Figure 4(a)-(b), the compiler-inserted nop instruction (`r0 = r0`) should read from physical register `33` (not `22`), which holds the value of `r0` before the PMEG. Since it is *guaranteed* that one and only one of these two instructions (`r0 = -r0` or `r0 = r0`) will execute and write to physical register `22`, downstream dependent instructions are completely oblivious that physical register `22` is written by a guarded instruction (Figure 4(e)).

As with *stall-at-issue*, after a PMEG is renamed, it is sent to the issue unit where it will wait until its guarding predicate condition is resolved. Once resolved, the instruction in the PMEG that is predicated-off will be discarded, and the predicated-on instruction will behave as if it were unpredicated.

The PMEGs approach could be done in a hardware-only manner by requiring the renaming logic to automatically inject register move operations in the back-end. Additional logic would have to be added to insert $\mu$-op move instructions: the logic would have to determine what type of move instruction to insert, and would also have to be able to split an instruction that writes to multiple destinations. Moreover, a hardware-only implementation would require that *every* predicated instruction be paired with a register move operation. Our compiler-driven approach has a much better view of predicated regions, and can therefore efficiently overlap instructions from mutually exclusive paths. We have found that the *if-then-else* statements in Mediabench, Biobench, and Spec 2006 write to at least one of the same registers around 90% of the time, and in the worst case this approach only doubles the number of instructions in each predicated region. Our experimental results in Section 6 show that increased region sizes are not problematic: the PMEGs approach is highly competitive with the idealized implementations of *stall-at-commit* and *stall-at-issue*.

| Feature | cmove | stall@ issue | stall@ commit | pmegs |
|---|---|---|---|---|
| Instruction decoder | ✓ | ✓ | ✓ | ✓ |
| Inject $\mu$-ops. | ✓† | ✓ | ✓ | |
| Rename table | | ✓ | ✓ | |
| Issue queue entries‡ | | ✓ | ✓ | |
| Renaming logic | | ✓ | ✓ | ✓ |
| Issue logic | | ✓ | ✓ | ✓ |
| Commit logic | | | ✓ | |

**Table 1. High level design modification summary.**
**Notes:** †Because conditional move requires three operands, some implementations split the instruction into two $\mu$-ops rather than require an extra operand network in the pipeline [11]. ‡Less aggressive implementations can achieve the same effect — but with worse performance — by injecting $\mu$-ops.

## 4.4 Summary of Design Points

This section concludes by summarizing the main design modifications that must be made to support lightweight predication. As Table 1 shows, all of the techniques that we consider in this paper need to modify the instruction decoder to identify predicate-defining instructions. All of the techniques except for `pmegs` require the renamer to inject $\mu$-ops. *Stall-at-issue* and *stall-at-commit* augment both the rename table and the renaming logic to keep track of conditionally-written registers. Instructions with multiple destinations may require extra support for *stall-at-issue* and *stall-at-rename*.

The `pmegs` approach requires additional logic in the renamer to simultaneously rename a PMEG group, and as with *stall-at-issue*, the issue logic must be augmented to discard predicated-off instructions. All techniques other than conditional move require support to snoop for resolved predicates.

# 5 Methodology

This section discusses the benchmarks, compiler, and simulator we use to collect performance results.

## 5.1 Benchmark Selection

Though `guard` can be used to predicate floating point operations, the general dearth of complex control flow in floating-point-intensive applications led us to consider "integer" applications. In this paper we consider the full Biobench [1] suite, all but one Mediabench [15] benchmark, and all but one Spec2006 [24] benchmark. We do not include *ghostscript* from Mediabench because the build process is too cumbersome, and GCC 4.1.2 cannot compile *483.xalancbmk* from Spec2006 at high levels of optimization. We use the standard inputs for Mediabench, and reduced-size inputs for Biobench and Spec2006.

| Feature | Description |
|---|---|
| Core frequency | 5.0 GHz |
| Pipeline stages | 10 Front-end, 8 back-end |
| Physical registers | 80 GPR, 80 FPR, |
| Fetch width | 6 |
| Commit width | 6 |
| Issue width | 8 wide |
| Functional units | 2 FXUs, 2 FPUs, 2 LD/ST, 1 BR, 1 CR |
| Issue queue | 60 entries |
| Reorder buffer | 100 entries |
| Instruction cache | 64KB, 4-way |
| L1 data cache | 64KB, 8-way |
| L2 cache | 4MB, 8-way |
| L3 cache | 32MB, 8-way |
| Memory | 1.33GHz DDR3 |
| Branch predictor | Combined gshare & local (see [22]) |

**Table 2. System parameters.**

### 5.2 Compilation

We use GCC version 4.1.2 and binutils 2.16.1. With this infrastructure, we are able to compile our benchmarks without any hand modification. We cross compile all of our applications on a Power5 system, then package the binaries and import them into our full-system simulator. We compile all benchmarks at optimization level "`-O3`".

### 5.3 Simulation Environment

We use the Mambo cycle-accurate full system simulator [4] — which has been vigorously validated against PowerPC hardware — as our evaluation infrastructure. We extend the validated model to a projected future system with the aforementioned predicated execution support. Table 2 lists some of the key parameters of the simulated baseline system. Of particular importance for this study is the branch predictor that we simulate. For our experiments we use the aggressive branch predictor featured in the IBM Power 5+ processors. The predictor uses three branch history tables: one for a *gshare* predictor with 16K entries, another for a local predictor with 8K entries, and a select predictor with 8K entries to predict which of the other two predictors is more likely to determine the correct direction. Please see [22] for complete details. In addition to the base system, we perform sensitivity studies by varying processor frequency, pipeline length, and pipeline width.

## 6 Results

This section evaluates the effectiveness of our PMEGs support by comparing it against `isel`, *stall-at-issue*, and *stall-at-commit*. We first present static results that give us insight into the opportunities available to the compiler. Table 3 lists the number of static regions that the compiler is able to predicate for each of our benchmark suites. The first, second, and third columns show the number of opportunities the compiler found for `isel`, `guard`, and `pmegs` respectively. The flexibility of `guard` and `pmegs` allows

| Suite | isel | guard | pmegs |
|---|---|---|---|
| Spec2006 | 1600 | 9981 | 8575 |
| Biobench | 1173 | 12434 | 11535 |
| Mediabench | 1924 | 5788 | 4840 |
| **Total** | **4697** | **28203** | **24950** |

**Table 3. Static predication opportunities.**

| Overlap | 0 | 1 | 2 | 3 | 4 | 5 | 6,7 |
|---|---|---|---|---|---|---|---|
| **Mediabench** | 12% | 52% | 16% | 13% | 3% | 4% | 0% |
| **Biobench** | 12% | 36% | 36% | 9% | 4% | 1% | 0% |
| **Spec** | 6% | 56% | 16% | 17% | 3% | 2% | 0% |

**Table 4. Histogram of "overlap" for `pmegs`.**

the compiler to predicate over five times as many regions as with `isel`. For instance, in the Biobench application `hmmer` there are several `isel` opportunities that the compiler was not able identify; in particular there are a few key missed opportunities where the compiler cannot guarantee the safety of memory accesses [20, 21].

Across the three benchmark suites we consider, 86% of the guardable regions are small enough to fit within eight instructions. This is a welcome result because we empirically determined that predicating regions beyond seven instructions typically worsens the performance of both `guard` and `pmegs`. For the *if-then-else* regions, Table 4 shows the distribution of the number of instructions from the *then* path that the compiler was able to match with instructions in the *else* path (to form a MEG). The column labeled "0" corresponds to cases where the compiler cannot find a single overlapping instruction. We see that the compiler is able to overlap at least one instruction — often significantly more — the vast majority of the time. With over one quarter of the regions being *if-then-else* regions, the ability to overlap instructions is important both for expanding the number of predicatable *if-then-else* regions, and for improving the performance of the resultant regions.

We now look at dynamic metrics and overall speedup data. The baseline configuration against which we compare our results is a projected future 5Ghz Power processor that is 8-way issue with 18 pipeline stages. We use the processor extensions described in Section 4 for *stall-at-issue*, *stall-at-commit*, and `pmegs`.

Figure 5 shows the performance attained with the various approaches described in this paper. The benchmarks are on the x-axis, and they are clustered according to benchmark suite: the 11 benchmarks on the left are from Spec2006, the 17 benchmarks in the middle are from Mediabench, and the seven benchmarks on the right are from Biobench. The y-axis shows the speedup over an optimized baseline (compiled with `-O3`). There are four numbers for each benchmark: the performance of `isel`, `guard` with *stall-at-issue*, `guard` with *stall-at-commit*, and `pmegs` (which stalls at issue). Table 5 shows the geometric mean speedup for each approach across our three benchmark suites.

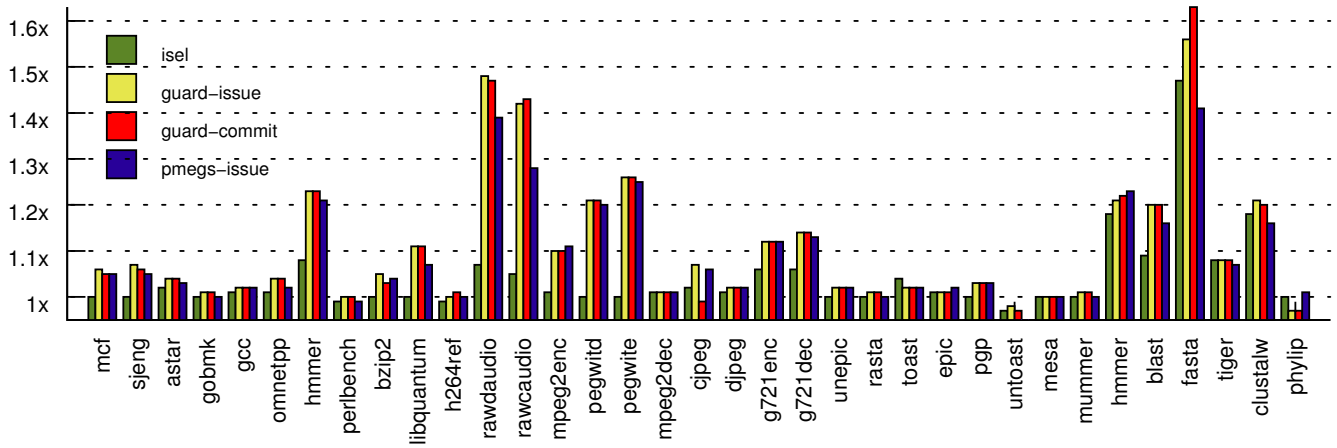


**Figure 5. Performance results for the baseline configuration listed in Table 2 (best viewed in color).**

| Suite | isel | guard-issue | guard-commit | pmegs |
|---|---|---|---|---|
| Spec2006 | 1% | 5% | 5% | 4% |
| Mediabench | 1% | 9% | 8% | 7% |
| Biobench | 16% | 19% | 20% | 16% |
| **Aggregate** | **3%** | **9%** | **9%** | **8%** |

**Table 5. Speedups over a -O3 compiled baseline.**

We see that across the benchmark suites `pmegs` solidly outperforms `isel`. In particular, for Spec2006 and Mediabench `isel` manages only a 1% performance improvement, largely because conditional move predication is difficult for a compiler to target. There are several highly-profitable `isel` opportunities in `hmmer` and `blast` that the compiler misses due to memory operations for which the compiler cannot determine memory safety [20, 21].

We also point out that `pmegs` nearly matches the performance of the `guard` approaches; we again remind the reader that the `guard` support is idealistic when compared to the work of [26]: the augmentation of the issue queue precludes inserting many conditional moves, and the `csel` conditional moves that are inserted execute in zero cycles. Thus, we can view the `guard` results presented here as an upper bound on performance for *stall-at-issue* and *stall-at-commit* support. Because the `pmegs` and `isel` approaches explicitly insert ISA moves and conditional moves respectively — which our simulator faithfully represents — they should not be able to match the idealized performance of `guard` presented here.

Within each benchmark suite in Figure 5, the benchmarks are sorted by branch predictability (from least to most predictable). Intuitively, the benchmarks with the worst predictability stand to benefit the most from the forms of predication this paper considers. Indeed we see `guard` and `pmegs` are able to substantially improve the performance of hard-to-predict applications such as `rawcaudio` and `rawdaudio`. However, there are also cases where predication is able to speedup relatively predictable applications (*e.g.*, `462.libquantum`, `g721encode`, `g721decode`, `pgp`, `clustalw`). A small subset of the benchmarks, namely `rasta` and `mesa` have very few dynamic opportunities for predication, and thus the performance is stable across all configurations.

The Mediabench benchmark `untoast` is an outlier in this graph— it is the only application for which `pmegs` appreciably slows down performance (∼9%). Roughly half of `untoast`'s cycles are spent in a method in which there are many guardable *if-then* branches. However, all of these branches are nearly perfectly predictable, and in the baseline case, the compiler has arranged the basic blocks such that all of the non-loop branches fall through to the next block. In essence, this layout forms a highly predictable trace, and predication only serves to serialize some of the computation therein.

Sachdeva et al. show that conditional move instructions are well-suited to the Biobench [1] suite of applications [21]. Many of the most frequently executed branches in these codes are simple *maximum* functions, which `isel` can efficiently implement. We see that indeed, `isel` is quite effective for this suite, matching the performance of `pmegs` at 16% speedup, yet falls short of `guard-issue`, which is at 19%, and `guard-commit` at 20%.

Though the *stall-at-commit* approach is more aggressive than *stall-at-issue*, in practice the technique provides only negligible performance benefits. This is consistent with [26], where the authors show that the "predicate slip" technique provides a roughly 1% benefit over the "select $\mu$-op" technique, which is comparable to our *stall-at-issue*. Notice that with the *stall-at-commit* approach, wrong-path instructions can cause performance degradation (e.g. `cjpeg`) since resources consumed by wrong-path instructions, including functional units and load-store queue entries, negatively affect the completion of "younger", useful instructions.

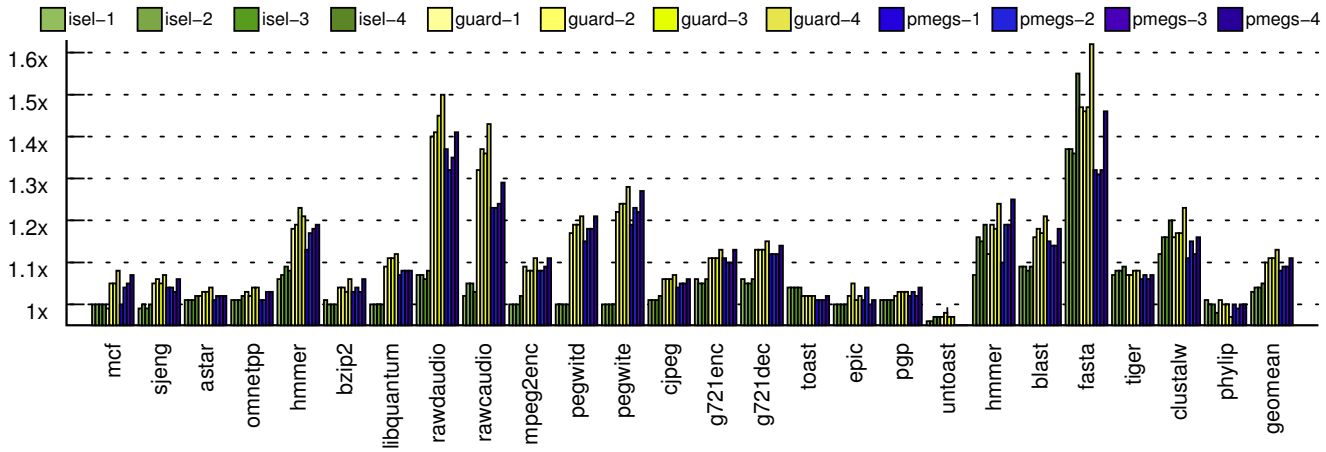


**Figure 6. Sensitivity analysis (best viewed in color).**

We also evaluate the effectiveness of feedback directed optimizations (FDO) for improving predication decisions. FDO adds an extra step to the compilation process: before production compilation, the developer must profile the application on a set of training data. The compiler then uses the collected profile — which provides knowledge about the application's run-time tendencies — to improve compilation decisions. We extended GCC's profiling infrastructure to profile for branch predictability.

With FDO, the compiler is able to improve performance for a few of the more unpredictable benchmarks. For `hmmer`, FDO markedly improves performance (1.45x *vs*. 1.23x for `pmegs`) by reordering basic blocks and precluding predictable branches from if-conversion. However, most of this gain is from reordering basic blocks, as FDO improves the performance of the baseline (*i.e.*, FDO without predication) by 16%. For `rawcaudio` and `rawdaudio`, on the other hand, even though FDO actually slightly decreases the performance of the baseline, the compiler is able to use the profile information to increase the performance of `pmegs` by 6% and 8% respectively. While select benchmarks show the promise of FDO, overall, the increases for `isel`, `guard`, and `pmegs` are commensurate with the increases attained for the baseline (an additional $\sim$1% increase in performance). Nonetheless, this should not be seen as a referendum on the usefulness of FDO: our predication heuristic for FDO is simplistic, and we do not yet fully understand the complicated interactions between the various passes enabled by FDO. Future work will consider bolstering our feedback directed optimization support.

### 6.1 Sensitivity Study

In this section we perform a sensitivity analysis to better understand the behavior of guarded regions. We consider two primary processor features: increasing the pipeline depth to simulate increased core frequency, and varying the issue width. All other parameters in Table 2 remain fixed. Figure 6 shows these results for most of our benchmarks; we omit the benchmarks for which there are few predication opportunities. The benchmarks are on the x-axis and the y-axis represents the speedup over a `-O3`-optimized baseline. The figure tracks four different configurations for each predication scheme: 1) 4Ghz, 4-wide; 2) 4Ghz, 8-wide; 3) 6Ghz, 4-wide; and 4) 6Ghz, 8-wide. The 4Ghz processor has 9 front-end and 7 back-end stages, while the 6Ghz processor has 13 front-end and 9 back-end stages.

We can see that as we increase the aggressiveness of the processor, the benefits of `guard` with *stall-at-issue* and `pmegs` both increase. However, the performance difference between `guard` and `pmegs` remains fixed. It makes sense that increasing the pipeline depth would increase the performance of the predication schemes: as the pipeline depth increases, so does the overhead of branch mispredictions.

Because `pmegs` tracks the optimistic performance of `guard` with *stall-at-issue* as we throttle the processor's aggressiveness, we believe that `pmegs` would be an attractive approach to enable predication support in the future.

## 7 Conclusion

Predication support necessarily complicates OOO processor design. This paper presents an approach called Predicated Mutually Exclusive Groups (PMEGs). While PMEGs requires compiler support, we show that a compiler can feasibly generate quality code for such a target: the `pmeg` extension we present achieves a geometric mean speedup of 8% across three popular benchmark suites. Moreover, it improves the performance of seven benchmarks by over 20%, and achieves a maximum speedup of 41%. We expect more sophisticated compiler heuristics to only improve upon these marks.

Our PMEGs approach comes within 1% of two optimistic implementations of hammock predication: *stall-at-*

*issue* and *stall-at-commit*. One of the reasons that the PMEGs approach is competitive with these techniques is that it is able to effectively "overlap" instructions from mutually exclusive paths, and it thereby minimizes the amount of "useless" nop moves it otherwise would have to insert. Finally, because the compiler presents a structured form of predication to the hardware, PMEG support requires fewer pipeline modifications than does prior art.

## 8 Acknowledgements

We would like to thank the various reviewers of this paper for their excellent feedback. We are also grateful for the support that we received from the members of the Novel Systems Architecture group and the Performance and Tools group at IBM's Austin Research Laboratory. This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.

## References

[1] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C. W. Tseng, and D. Yeung. Biobench: A benchmark suite of bionformatics applications. *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 2–9, 2005.

[2] D. August, W.-M. Hwu, and S. Mahlke. A Framework for Balancing Control Flow and Predication. In *International Symposium on Microarchitecture*, pages 92–103, 1997.

[3] D. Bhandarkar. *Alpha Implementation and Architecture*. Digital Press, 1996.

[4] P. Bohrer, M. Elnozahy, A. Gheith, C. Lefurgy, T. Nakra, J. Peterson, R. Rajamony, R. Rockhold, H. Shafi, R. Simpson, E. Speight, K. Sudeep, E. V. Hensbergen, and L. Zhang. Mambo - A Full System Simulator for the PowerPC Architecture. *ACM SIGMETRICS Performance Evaluation Review*, 31(4), March 2004.

[5] W. Chuang and B. Calder. Predicate Prediction for Efficient Out-of-Order Execution. *In Proceedings of the 17th Annual International Conference on Supercomputing (ICS)*, pages 183–192, 2003.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 2001.

[7] A. Darsch and A. Seznec. Out-of-Order Predicated Execution with Translation Register Buffer. Technical Report N°5011, Institut National de Recherche en Informatique et en Automatique, November 2003.

[8] The GNU Project. *http://gcc.gnu.org*.

[9] T. Heil and J. Smith. Selective Dual Path Execution. November 1996.

[10] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual: Volume 1: Basic Architecture*, 2008.

[11] R. E. Kestler. The Alpha 21264 Microprocessor. *IEEE Micro*, pages 24–36, 1999.

[12] H. Kim, J. A. Joao, O. Mutlu, and Y. N. Patt. Diverge-Merge Processor: Generalized and Energy-Efficient Dynamic Predication. *IEEE Micro*, 27(1):94–104, 2007.

[13] A. Klauser, T. Austin, D. Grunwald, and B. Calder. Dynamic Hammock Predication for Non-Predicated Instruction Set Architectures. In *PACT '98: Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, page 278, Washington, DC, USA, 1998. IEEE Computer Society.

[14] A. Klauser, A. Paithankar, and D. Grunwald. Selective Eager Execution on the PolyPath Architecture. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[15] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems. In *International Symposium on Microarchitecture*, volume 30, pages 330–335, 1997.

[16] S. A. Mahlke. *Exploiting Instruction Level Parallelism in the Presence of Branches*. PhD thesis, University of Illinois at Urbana-Champaign, Department of Electrical and Computer Engineering, 1996.

[17] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W.-M. W. Hwu. A Comparison of Full and Partial Predicated Execution Support for ILP Processors. *SIGARCH Comput. Archit. News*, 23(2):138–150, 1995.

[18] S. A. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *International Symposium on Microarchitecture*, volume 25, pages 45–54, 1992.

[19] D. N. Pnevmatikatos and G. S. Sohi. Guarded Execution and Branch Prediction in Dynamic ILP Processors. *SIGARCH Comput. Archit. News*, 22(2):120–129, 1994.

[20] P. Ratanaworabhan and M. Burtscher. Load Instruction Characterization and Acceleration of the BioPerf Programs. In *IEEE International Symposium on Workload Characterization*, pages 71–79, October 2006.

[21] V. Sachdeva, E. Speight, M. Stephenson, and L. Chen. Characterizing and Improving the Performance of Bioinformatics Workloads on the Power5 Architecture. In *International Symposium on Workload Characterization*, Boston, MA, September 2007.

[22] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. Power5 System Microarchitecture. *IBM Journal of Research and Development*, 49(4-5), 2005.

[23] A. Smith, R. Nagarajan, K. Sankaralingam, R. McDonald, D. Burger, S. W. Keckler, and K. S. McKinley. Dataflow Predication. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 89–100, December 2006.

[24] SPEC.org. *http://www.spec.org*.

[25] E. Sprangle and Y. Patt. Facilitating Superscalar Processing via a Combined Static/Dynamic Register Renaming Scheme. In *MICRO 27: Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 143–147, New York, NY, USA, 1994. ACM.

[26] P. H. Wang, H. Wang, R.-M. Kling, K. Ramakrishnan, and J. P. Shen. Register Renaming and Scheduling for Dynamic Execution of Predicated Code. In *HPCA*, pages 15–26, 2001.