Discerning the Dominant Out-of-Order Performance Advantage: Is it Speculation or Dynamism?

Daniel S. McFarlin
Carnegie Mellon University
dmcfarlin@cmu.edu

Charles Tucker Craig Zilles
University of Illinois at Urbana-Champaign
{cetucker,zilles}@illinois.edu

Abstract

In this paper, we set out to study the performance advantages of an Out-of-Order (OOO) processor relative to in-order processors with similar execution resources. In particular, we try to tease apart the performance contributions from two sources: the improved schedules enabled by OOO hardware speculation support and its ability to generate different schedules on different occurrences of the same instructions based on operand and functional unit availability. We find that the ability to express good static schedules achieves the bulk of the speedup resulting from OOO. Specifically, of the 53% speedup achieved by OOO relative to a similarly provisioned in-order machine, we find that 88% of that speedup can be achieved by using a single "best" static schedule as suggested by observing an OOO schedule of the code. We discuss the ISA mechanisms that would be required to express these static schedules.

Furthermore, we find that the benefits of dynamism largely come from two kinds of events that influence the application's critical path: load instructions that miss in the cache only part of the time and branch mispredictions. We find that much of the benefit of OOO dynamism can be achieved by the potentially simpler task of addressing these two behaviors directly.

Categories and Subject Descriptors D.3.4 [**Software**]: Programming Languages–*Processors:Compilers, Optimization*, C.0 [**Computer Systems Organization**]: General–*Hardware/software interfaces*

General Terms Performance

Keywords Optimization; Speculation; Dynamic Scheduling

1. Introduction

The current state of parallel programming has unleashed a tension of design criteria on modern general-purpose CPUs. There are neither so few nor so many parallel programs that either parallel or single-thread performance can be marginalized. Instead CPU designers must design chips with high single-thread performance, but vigilantly maximize performance/area and performance/watt.

One important design decision that plays centrally in this trade-off is the choice between in-order and out-of-order instruction scheduling by the hardware. GPUs and lower-end processors have demonstrated the performance/area and performance/watt advantages of in-order, but currently out-of-order (OOO) designs hold the mindshare for achieving peak single-thread performance.

We perceive, however, that there are two potential sources for the performance advantage between the hardware OOO schedules and compiler-generated schedules as executed by in-order machines:

1. the OOO has few constraints other than true dependencies and functional unit availability to constrain schedule generation, whereas existing ISAs prevent the expression of some *speculative* schedules by compilers.
2. OOO can execute the same instructions in different schedules at different points of the execution, reacting to the specific events observed during each execution. We'll call this advantage *dynamism*.

Little effort has been directed at quantifying the contributions of each of these effects, but we believe that this is an important question. Specifically, if the difference between OOO and in-order is largely not the result of dynamism, but rather due to the inability of the compiler to express the desired schedule given the available ISA interface, this is something that can be addressed by further ISA development.

To this end, this paper reports a study we undertook to characterize the relative importance of dynamism in OOO and better understand the schedule differences between in-order and OOO schedulers.

This paper makes the following contributions:

- We demonstrate that most (on average 88%) of the OOO's performance advantage is due to the speculation support that enables the formation of high quality schedules.
- We show that the OOO's remaining performance advantage from dynamism is predominantly from its ability to deploy different schedules in the immediate locus of a cache miss or branch misprediction.
- We quantify the ability of a complexity-effective, "off-the-shelf" solution for cache miss recovery coupled with a simple mechanism for misprediction recovery to cover the remaining performance gap to the OOO.
- We provide recommendations for realizing performance competitive in-order designs given current technology and workload trends.

The remainder of the paper is organized as follows: Section 2 elaborates on the need for the present work. Section 2.1 examines the motivation for the strong tendency towards OOO designs in industry and the factors limiting the performance competitiveness of in-order designs. In Section 2.2 we employ a criticality framework to

ASPLOS'13, March 16–20, 2013, Houston, Texas, USA.

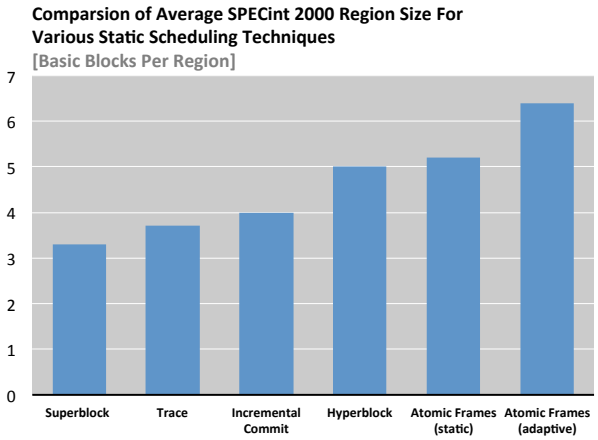

Figure 1. Average region size achieved by top-performing scheduling techniques for in-order machines.

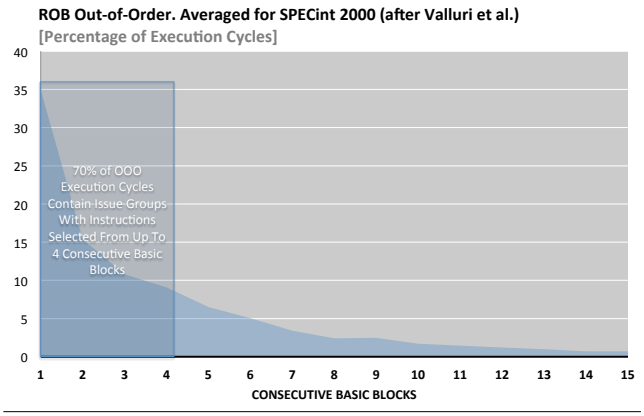

Figure 2. The basic block overlap ability of a representative out-of-order design.

explore the theoretical and practical implications of dynamism on performance and provide intuition as to its principle and we argue, limited advantages over the OOO's baseline speculation support. Section 3 looks at our methodology for quantifying the relative contributions of dynamism and speculation to OOO performance; the presentation and discussion of our results appear in Sections 4 and 5. The rich history of the in-order vs. OOO debate along with the challenges academia and industry encountered in trying to match the OOO's performance are explored in Section 6. We conclude in Section 7.

2. Background/Motivation

Previous work has observed that OOO scheduling hardware tends to spend most of its time scheduling the same small subset of the static program, and moreover tends to create the same cycle-by-cycle schedule repeatedly. Caching these schedules is the motivation for Execution Caching [47], to allow the scheduler to be shut down (which they argue produces power savings) some of the time.

Our work extends this intuition in several ways. First, previous work has been in terms of actual architectural implementations, which tends to mask the theoretical limitations of static scheduling with implementation details. We seek conclusions applicable more generally to statically scheduled architectures. In addition, we want to tease apart the OOO performance advantage to see whether an aggressive in-order design has any hope of matching its performance, and what sorts of features it would require to do so.

2.1 The Perceived Out-of-Order Performance Advantage

Despite some of the well-known advantages offered by in-order designs (usually lower power, higher frequency, and reduced HW complexity), Out-of-Order designs dominate general purpose computing and are proliferating into domains where OOO designs were conventionally seen as poorly suited e.g. transaction processing [41], data/control-plane processing [20] and the embedded space [16]. While direct comparisons between in-order and OOO designs are difficult, OOO designs seem to consistently provide higher single-thread performance relative to comparably provisioned in-order designs [22].

The common explanation for the IO/OOO performance disparity is that the OOO is inherently better at exploiting memory-level parallelism as even the most sophisticated static schedule is hobbled by the stall-on-use/head-of-line blocking problem inherent to cache misses in IO designs [2, 22]. In contrast, the enduring trend over several generations of OOO designs is their ability to tolerate an ever increasing number of longer latency cache misses; Intel's 22nm Ivy Bridge boasts a window size nearly twice that of the 45nm Penryn [24]. Indeed, given the ever increasing complexity and non-determinism of the memory hierarchy, the MLP advantage seems to be what is motivating the trend towards OOO designs. Recent in-order processors have explicitly incorporated HW features to "buy back" [28] this loss of MLP relative to the OOO including run-ahead execution (Power6) and Subordinate scout threading (Rock) [22]. Yet, a recent in-order design that combined a sophisticated run-ahead scheme, advanced prefetchers, register checkpointing, and gated store-buffer with a good compiler was still some 42% slower than a comparably provisioned OOO design [22]. Why?

Figures 1 and 2 highlight a major challenge for in-order designs in matching the OOO: the OOO's ability to simultaneously execute instructions from a consistently larger number of consecutive basic blocks. Figure 1 compares the average region size (number of basic blocks) found by some of the best performing static scheduling techniques documented in the literature when applied to SPECint 2000. Trace Scheduling [42], Superblock [21] and Hyperblock [17, 37] are all well known, often profile-driven global scheduling techniques employed by static compilers. Incremental Commit/Non-Atomic Traces [51] and Atomic Frames [36] are techniques employed by runtimes to dynamically optimize binaries; the first three techniques require modest amounts of HW support; the last two require fairly aggressive HW implementations with the adaptive version of Atomic Regions continually re-optimizing certain regions based on performance/event counters.

Excluding Atomic Frames, the first four scheduling techniques produce regions of, on average, between 3.3 and 5 basic blocks in size. How does this compare to the OOO? Figure 2 shows that overlapping the execution of instructions from 4 basic blocks is only sufficient to account for about 70% of the OOO's execution cycles; the OOO spends the remaining 30% of its execution cycles issuing instructions derived from 5 or more consecutive basic blocks. Note Figure 2's "long tail"; even overlapping the execution of instructions from 6 consecutive basic blocks, comparable to the overlap achieved by the best performing Atomic Frames variant, only accounts for about 82% of the OOO's execution cycles.[1] Adding yet another consecutive basic block (a region size increase of 16%) contributes only another 3% improvement to coverage. Furthermore, the emerging performance trend of static scheduling

[1] Architectures such as TRIPS [17] and the Region Slip variant of rePlay [45] provide for the overlapping of Hyperblocks and Atomic Frames, respectively, at the cost of substantially increased hardware, runtime and compiler complexity as well as costlier misspeculation penalties

techniques for SPECint 2006 is not particularly encouraging; [43] reports that the average region size for global trace scheduling fell about 15%, from 3.7 basic blocks to 3.2 basic blocks.

While this data re-affirms the well-known need for hardware support for software speculation, state-of-the-art static scheduling techniques that rely on such support still fail to match the dynamic schedules produced by the OOO. We argue that the primary enabler for the higher quality, dynamic schedules is the OOO's more general and better provisioned speculation support facilities (very large ROB, renamer, larger Load/Store Queue, dynamic/adaptive memory disambiguation) [49] and the generally lower cost of misspeculation; the OOO can more readily adapt to data misspeculation [10] while its finer-grained commit and cheaper recovery mechanisms give it an edge over Non-Atomic Traces and Atomic Frames [9, 45]. Consequently, a combination of HW support for software speculation, some form of non-blocking instruction issue and, as will be demonstrated, the ability to recover from mispredicts is necessary to close the performance gap. To the best of our knowledge, no industrial or academic design has explored the theoretical desirability and practical necessity of this combination.

2.2 Scheduling vs. Dynamism

It is easy to attribute the benefits of OOO almost entirely to dynamism, that is the ability of OOO to generate schedules at runtime based on the actual path taken by the code and execution latencies of the operations. In this section, we attempt to tease apart what OOO provides to understand which component could potentially be provided by simpler mechanisms.

Path Specificity:

Because out-of-order performs scheduling after branch prediction it gets the benefit of scheduling based on the predicted path. This creates a important practical distinction between out-of-order and in-order when the predictability of a branch exceeds the branch's bias. Consider, for example, the following code:

Block A is terminated by a branch that can flow to either block B or block C. Each basic block contains a collection of instructions with dependencies, which result in the schedule for each basic block to be narrower than the machine's width. There is, however, potential for overlap between basic blocks (*i.e.*, the critical path through the whole application is not the sequence of critical paths through each basic block).

On an out-of-order machine, if block B is predicted to follow block A, then the instructions from block B will be intermingled with those from block A. Similarly, if the path from block A to block C is predicted, then C's instructions will be intermingled with block A.

Of course, many compilers are capable of hoisting instructions across basic block boundaries, often using ILP scheduling techniques like superblock scheduling. What ILP scheduling techniques accomplish is to statically pick a path for optimization (*e.g.*, the path from block A to B) and hoist instructions from block B into block A. Using this technique, when the code executes the path from A to B, it can achieve schedule quality similar to that of the out-of-order machine. When the path from A to C is taken, however, the collection of instructions from block B that were hoisted to block A are wasted work and we have achieved no overlap between C and A [2]. This is not a serious problem if the the path from A to B is much more frequent than the path from A to C. Furthermore, the behavior of the A to C path with an ILP scheduling compiler is much like the behavior of the out-of-order machine when it predicts incorrectly to follow the path from A to B when the path A to C is needed; in this case, the out-of-order will hoist much of B before the branch in A is resolved at which point C will need to be fetched, likely resulting in no overlap between A and C.

In short, an ILP compiler typically achieves good overlap in proportion with the bias of the branch, while the out-of-order does so in proportion to the predictability of the branch. This results in a significant advantage for the out-of-order because branch predictability is almost always higher than branch bias, sometimes significantly so.

Variable Latency:

Tolerance of long latency operations is an often touted strength of out-of-order processors, but it is important to recognize that static scheduling — through techniques like modulo scheduling — can effectively tolerate long latencies as well. The key advantage of out-of-order is two-fold: first, in its ability to tolerating variable latency, and, second, in how this interacts with path specificity.

While most instructions on modern machines have a fixed latency, some instructions, notably load instructions, have a variable latency. This variable latency is important when it potentially affects the critical path. Consider the following three scenarios:

Scenario A: (off the critical path)

```
(int *)A[N];
int C[N];

for (int i = 0 ; i < N ; i ++) {
  C[i] = *A[i];
}
```

This code exhibits a "spine and ribs" structure where the loop carried dependence (`i++`) forms the critical path and the work of the loop (a serial dependence chain of two loads and store) are a rib hanging off that spine. Because the variable latency — the second load — is on the rib (and hence off the critical path) a compiler for an in-order machine could theoretically schedule the consumers of that load for its worst case latency. This would achieve the same performance as the out-of-order, because there is little benefit to completing the spines early.

Scenario B: (on the critical path)

```
int A[N], index = 0;

for (int i = 0 ; i < N ; i ++) {
  index = A[index];
}
```

In this code example, we can pack the code as tight as possible. The critical path will always go through the loads, no matter what their latency.

Scenario C: (the critical path is a function of latency)

What is particularly challenging for static scheduling to deal with is when the critical path through a region is a function of whether an access hits or misses in the cache. Below is a simple, illustrative example of this consisting of 4 loads. The data dependence graph for this code is shown in Figure 3(a).

```
z = p1->x->z + p2->x->z;
```

In this code, how the second pair or loads should be scheduled depends on which loads miss, so that we can overlap misses. Figure 3(b) shows a scenario where the `p1->x->z` load should be scheduled last so that both misses overlap, and Figure 3(c) shows the complementary scenario that benefits from the other ordering. This is a situation where there is no substitute for dynamism.

Early Branch Resolution:

[2] Alternatively, we could hoist some instructions from both B and C into block A, but typically machine width and functional unit limitations prevent us from hoist as many instructions from either path if we choose to hoist from both.

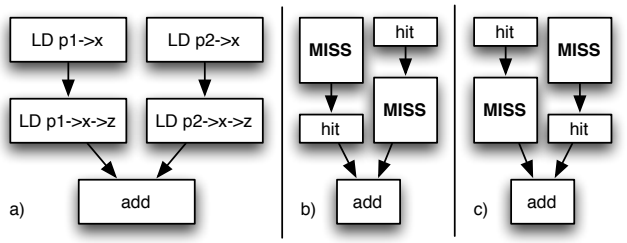

Figure 3. Scenario C variable latency code example: a) data dependence graph, b) `p1->x` and `p2->x->z` miss, c) `p2->x` and `p1->x->z` miss.

While it doesn't happen frequently, an OOO machine has the potential to overlap branch misprediction latency with other execution. In the code that follows, the branch on `b` does not depend on the computation before it, so it can be resolved in parallel with the computation.

```
p->y ++;
if (b) {
    ...
```

Achieving this overlap can be challenging for in-order machines because branches generally must be placed at the end of their basic block (to demarcate which instructions belong in the block), which also constrains their place in the schedule. Two solutions to this problem are available. On the software side, the pre-branch computation can be pushed down onto both downstream paths at the cost of code replications. On the hardware side, "prepare-to-branch" mechanisms [29] have been proposed that separate the computation of a branch outcome from the branch's role as a basic block boundary; IBM's Cell processor is a recent example employing such a mechanism [19]

Branch Misprediction:

One of the fundamental advantages of dynamic scheduling over static schedules is that ability of the dynamic scheduler to "ramp up" after and during a branch mispredict. Specifically, the dynamic scheduler has superior recovery from branch mispredicts as it is able to respond to actual instruction readiness that occurs during and after the mispredict recovery period rather than the readiness assumptions encoded in a static schedule. The main insight here is that the ILP instructions hoisted by the static scheduler between a critical path producer and consumer to cover non-unit latencies can in fact extend the critical path when these non-unit latencies have already been covered (in part or in full) by the misprediction recovery latency. Once the correct path instructions hit the execute stage, a statically scheduled machine must still churn through the non-critical hoisted instructions before reaching the critical path consumer despite the consumer's immediate readiness upon misprediction recovery; a dynamically scheduled machine has no such limitation and can immediately execute the critical path consumer.

We illustrate this phenomenon through two examples excerpted from SPEC2000 Integer benchmarks and visualized in Figures 4 and 5. Each figure follows the same convention:

- Time flows from left to right
- Instructions that issue in the same cycle are shown in the same column
- Non critical path instructions are denoted by a label of the form `ILP[0-9]+`
- **A** shows the in-order, linear instruction sequence as emitted by the compiler; Critical path instructions are identified by decorated borders
- **B** shows an execution trace of the dynamically scheduled code (OOO) when the fall-through is correctly predicted
- **C** shows an execution trace of the statically scheduled code (VLIW with packets shown as rounded rectangles) when the fall-through is correctly predicted
- **D** shows an execution trace of the dynamically scheduled code (OOO) when the fall-through is mispredicted
- **E** shows an execution trace of the statically scheduled code (VLIW) when the fall-through is mispredicted

All examples have consumers of loads located in the fall-through path of a conditional branch. Both the dynamic scheduler and static scheduler are able to cover the load-to-use latency of the loads by hoisting instructions from the fall-through-path after the branch but before the consumer. In the common case of a correct branch prediction (not-taken), the dynamic schedule and the static schedule execute in the same number of cycles. The real disparity in performance comes in the event of a branch misprediction. The load prior to the branch executes in the same cycle in both schedules. Assume that both redirect the front-end at the same time and both incur the same mispredict penalty. In both cases, the L1 hit latency is covered by the time to redirect the front-end. In other words, for both scheduling techniques, the consumer of the load is ready to execute as soon as it enters the window. In the parlance of Fields criticality [14], the consumer of the load issued prior to the mispredicted branch is fetch critical.

The delinquent execution of `r0` in **E** relative to **D** in Figure 4 (derived from `gap`) shows the performance difference between the two scheduling techniques; the dynamic scheduler is able to immediately execute the consumer of `LD(A)` (it is the oldest, ready instruction in the window). In contrast, the static schedule that enters the window after the branch misprediction recovery consists of instructions that were hoisted to cover the L1-hit latency but this latency has already been covered by the mispredict latency. As a result, these hoisted instructions now prevent the immediate execution of the load's consumer.

Though the branch mispredict latency extends the critical path length for both the dynamically and statically scheduled versions, the hoisted instructions in the static schedule now further extend the critical path length by delaying the use of a critical, ready instruction. In other words, a static schedule to cover the L1-hit latency along the critical path is now penalized by the L1-hit latency. An extension of this scenario occurs when, with sufficient profiling, the static scheduler schedules for an L2 hit and is able to find sufficient ILP to cover the L2-hit latency. In this case, the execution of the use of the L2-hit is delayed by 10 cycles (the mispredict recovery latency) when the redirected instruction stream finally reaches the execute stage.

Figure 5 derived from `eon` demonstrates that this problem is compounded by the presence of cascading loads *i.e.*loads that feed other loads. In this example, `r0 = LD(A)` feeds `r1 = LD(r0)` before a conditional branch. The consumer of `r1` is after the conditional branch. As in example 1, in the case of a correctly predicted fall-through, the performance of the static schedule is identical to that of the dynamic schedule; the static schedule initiates `r0 = LD(A)` before the conditional branch and pushes the use of `r0(r1 = LD(r0))` beneath the branch. The L1-hit latency of `LD(A)` is covered by hoisted instructions; the L1-hit latency of `LD(r0)` is similarly covered. With an L1-hit latency of 4 cycles, the path to `r1` from `LD(A)` is 8 cycles. As in Example 1, both scheduling techniques experience a mispredict. For the OOO, `LD(A)` and `LD(r0)`

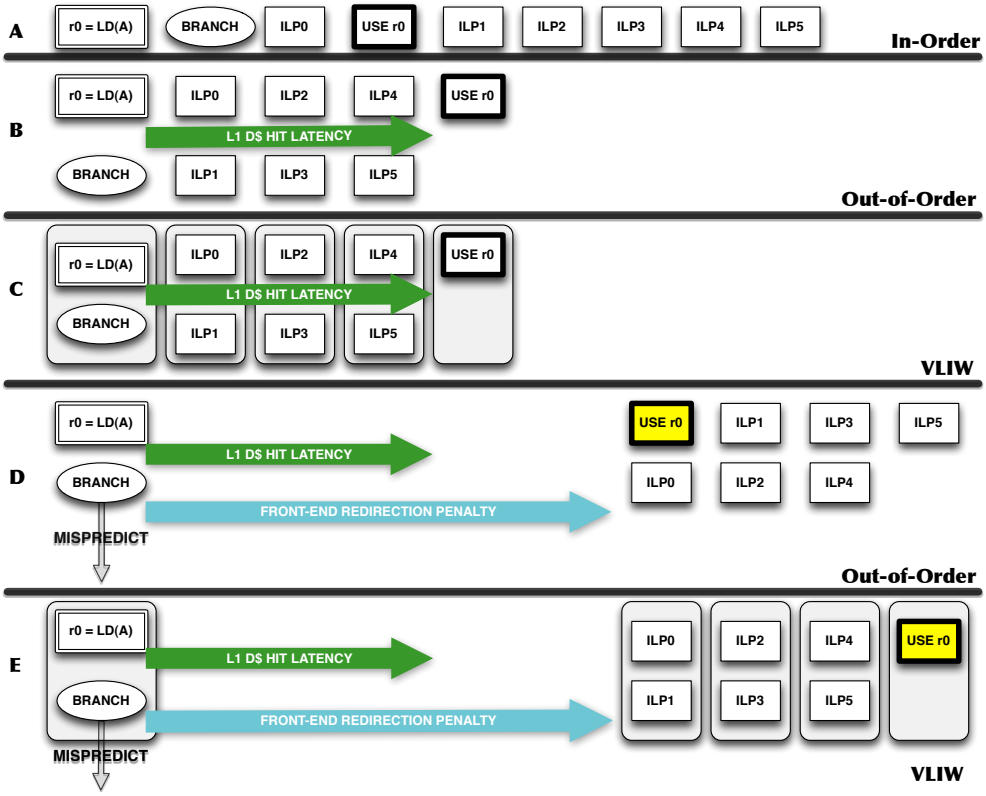

Figure 4. Misprediction recovery latency covers L1-hit latency

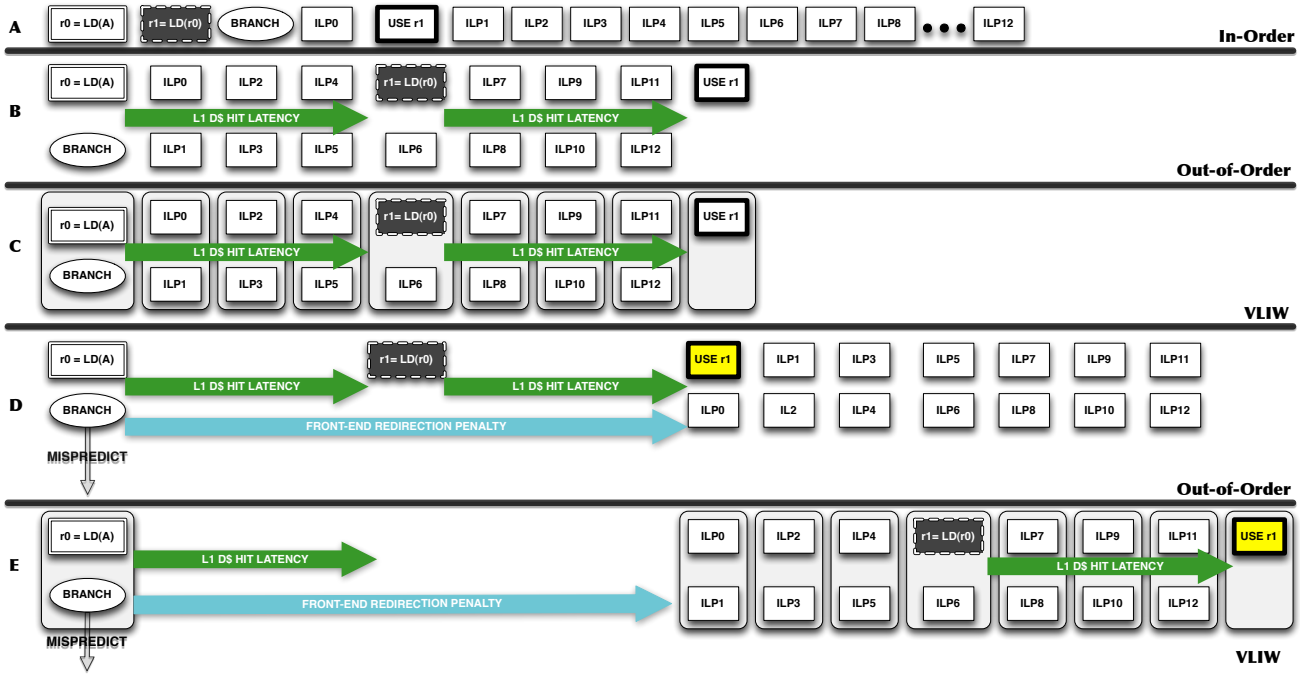

Figure 5. Misprediction recovery latency covers L1-hit latencies of a Load chain

can both execute in the shadow of the mispredict recovery. When the correct path instructions finally enter the window, `LD(r0)` , as the oldest, ready instruction executes immediately. In contrast, because the static schedule has pushed the `r1 = LD(r0) => USE r1` chain below the branch and hoisted ILP instructions to cover the L1-hit latencies for `r0 = LD(A)` and `r1 = LD(r0)` , the execution of `r1` is delayed by 8 cycles.

3. Experimental Method

In an attempt to separate the speculation and dynamism aspects of OOO execution, we have developed a simulator framework that allows us to run a benchmark on the simulator configured as an OOO processor and capture the dynamic schedules used, identify a dominant schedule for each region, and replay these schedules in a second in-order execution. In this section, we describe our choice of benchmarks (Section 3.1), how we collect the dynamic schedules (3.2), how we replay these schedules

It should be noted that this work is a limit study of sorts. Specifically, we make no attempt to describe the mechanisms that would be necessary to produce these schedules nor the architectural features that would be necessary to encode these schedules. We do, however, think that a useful mental model for the system under study is a machine like the Transmeta Crusoe [9], which consisted of a VLIW processor coupled with the CMS dynamic binary translator. Our study can be viewed as investigating the potential for a futuristic dynamic optimizer provided it was not constrained in the static schedules it could generate and it could effectively profile the execution so as to generate relevant schedules.

In addition to scheduling code, such a dynamic optimizer is capable of applying other optimizations (*e.g.*, converting a store-to-load forwarding into a register communication) that can reduce the program's critical path and/or number of executed operations. While these transformations are valuable and could potentially improve the performance of an optimized in-order execution past a traditional out-of-order, we will not consider those here because they would only make our goal of understanding the role dynamism plays in OOO execution by obfuscating where performance is coming from.

3.1 Benchmarks

Because the largest benefits of OOO have historically come from non-numeric programs[3], we focus on integer programs, from the SPECint 2000 benchmark suite, in this work. For our Alpha-based simulation framework, our executables are compiled using the DEC C Alpha compiler (V5.9- 005), with peak optimizations enabled for the 4-wide, in-order DEC Alpha 21164, but no profile feedback.

For each experiment, we run several checkpoints from each benchmark in SPECint 2000. Each run executes for 2 million instructions to warm up caches and branch predictors then execution time is measured for the next 10 million instructions.

3.2 Harvesting Schedules

Generating our static schedules is a two step process. In the first step, we run our OOO simulator and collect a trace of the schedule as it was scheduled by the OOO machine. To make the selection of a dominant schedule tractable, we find dominant schedules for various regions of code and then *stitch* them back together. Thus, as we collect the trace, we break it into *snippets* of code, terminating snippets at backwards branches and function returns. Each time a snippet is captured, we record the cycle that an instruction executed relative to the first instruction of each snippet (which executed in cycle 0 by definition). Due to the nature of the OOO, it is common for instructions to have negative offsets. We also record the PC and the offset of the first instruction of the next snippet, which we use as a *stitching point* when we attempt to put the snippets back together.

We aggregate all of the snippets from all of the checkpoints of a given benchmark and post-process them together. The first step is to discard any schedules that embed a mispredicted branch; we found that these schedules are too pessimistic to assume as our static schedule. Second, any time the stitching point has a negative offset (*e.g.*, when the first instruction of a subsequent snippet happens to execute before the first instruction of this schedule), we set the stitching offset to zero to avoid a deadlock in our simulations. After these two transformations, we simply select the most common schedule (*i.e.*, the mode) to be our static schedule. This schedule along with its *key* – the sequence of PCs in program order that make up the schedule – are entered into a "database" which is used in the subsequent simulation. Together these snippets provide more than 99.999% coverage of our simulated dynamic instruction stream.

As our "region formation" heuristic is very simple, it has some known shortcomings. First, it often results in short snippets. This is largely mitigated by overlapping the schedule of snippets through the stitching discussed below. Tight loops with high trip counts have the potential to be more problematic. Our methodology can underestimate the performance of such loops because it is constrained to produce code with integer initiation intervals. For example, a loop with 5 instructions will result in an initiation interval of 2, where a compiler, by unrolling this loop could pack 4 iterations into 5 cycles.

3.3 Replaying Dominant Schedules

We have modified the front end of our simulator to allow us to replay these dominant schedules as if they were static schedules provided by a compiler. At the fetch stage of our timing simulator, we exploit the functional-first nature of our timing simulator to examine the upcoming dynamic trace of instructions to generate a key and select the appropriate snippet. By construction, none of our snippet keys are a prefix of any other, so this lookup process is a simple one. In this way, we construct a short queue of snippets and their schedules.

We combine the snippet schedules into a composite schedule by placing each successive schedule's cycle 0 at the insertion point indicated by the previous schedule and populating the composite schedule. Because our snippet schedules are selected independently, there is the potential that we selected a snippet schedule that never executed with the schedule for the snippet before it. As a result, this naive stitching can result in unrealizable schedules due to oversubscribing the width of the machine or by violating data dependencies. If one of these violations is detected, the insertion point is "bumped" forward a cycle and the process is repeated. In principle, most of this work could be done offline, but as our post-processor doesn't actually decode the instructions and hence cannot detect dependence violations, it was simplest to implement this as part of the front end. Most notably, stitching is performed in the front end with no awareness of dynamic events like branch mispredictions or cache misses, so is performed the same way each time for a given sequence of basic blocks.

Our simulator framework lacks support for feeding wrong-path instructions into our timing model. When our predictor model indicates a mispredicted branch, we stall fetch of subsequent instructions until the branch hits an execute unit (and resolves), after which fetch begins again. We use the same approach for all simulations.

[3] Numeric programs often have extensive instruction-level parallelism, more regular control flow, and simpler memory-access patterns (*e.g.*, fewer pointer-based data structures) which make traditional compiler optimization more successful resulting in very competitive performance on in-order machines.

Front end	4-wide fetch (no I$ modeled)
	10-cycle branch mispredict
	GShare (16b index, 16b history)
	Cascading Indirect Predictor
	2K-entry target buffer
	16-entry RAS
Execution	Maximum 4 of: 4 Int (1 cycle simple, 5 mul/div) 2 FP (4 simple, 20 complex) 2 Load/Store (1 AGEN/TLB) + cache latency)
	64 entry window
Back end	128 entry ROB
	4-wide commit
L1$	32KiB, 64B line, 4-way LRU, 3 cycles
L2$	256KiB, 64B line, 8-way LRU, 25 cycles
DRAM	150 cycle latency, 16B/cycle
iCFP	64 entry slice buffer

Table 1. Simulated machine parameters

As previously noted our snippet coverage is almost but not quite 100%. So as to be not too pessimistic with the less than 0.00001% of the total instructions that aren't covered by snippets, dynamic scheduling is used for these instructions in the simulator. We don't, however, allow the execution of these instructions to overlap with snippets before or after them, which means our handling of these instructions is likely conservative.

3.4 Processor Model

Our processor model is configured to approximate the resources and latencies in modern OOO processors [34], as shown in Table 1.

In order to support pre-scheduled code in the simulator, we need to make a handful of simplifications and modifications to the processor model's front end. First, as previously noted, we use a functional-first methodology which lacks support for injecting wrong-path instructions into the execution. The impact of this limitation is mitigated by the fact that we are studying the relative performance of the pre-scheduled runs to a baseline (OOO) and all runs incur this limitation.

Second, it would be difficult to accurately model the instruction cache performance of the pre-scheduled runs, so we assume a perfect instruction cache. This has little impact on the baseline runs because our benchmarks have relatively small instruction working sets, but may result in some over-prediction of performance for the the pre-scheduled runs which will necessarily require some amount of code replication in order to achieve the desired static schedules in the presence of control flow.

Third, we perform branch prediction using the PC of branches as they appear in the original execution stream. In a real system, any place code has been replicated, each of those static instances would likely index into different places in the branch predictor. This separation can have negative (*e.g.*, longer training times, more predictor aliasing) and positive (*e.g.*, the streams may be more predictable when separated) affects. We cannot, however, consider this in our simulation infrastructure because our snippets selection is derived from future path information such that many snippet branches are completely biased. By using prediction based on the original program stream, we can ensure that our performance results are not benefiting from any prediction anomalies.

When branch mispredictions occur, their penalties are enforced at the point where the branch was scheduled; that is, post-branch cycles of the schedule are not fetched until the branch itself has executed. This does not however prevent right-path instructions that were scheduled above the branch to execute in parallel with the branch resolution as if they had been hoisted above the branch by the compiler. We are not, however, able to model the contention from wrong path instructions that the compiler might also have desired to hoist above the branch.

3.5 Avoiding Cache Miss-Induced Stalls

Continual Flow Pipelining [46] was originally proposed to give an existing OOO processor a larger effective window without growing complex structures. It does this by "slicing out" instructions transitively dependent on (and stalled by) a load miss (or equivalently any other high-latency event) into a separate queue and re-inserting them in the back-end when the load miss returns. In-order retire still requires that these sliced instructions hold ROB slots, but other resources (physical registers, reservation station slots) can be used for independent work. Notably, this does not allow for load/store queue entries to be reclaimed; dependencies through both registers and memory must be identified to slice out the correct instructions. Once the load miss comes back, the design "rallies", leveraging existing superscalar / SMT front-end hardware to dovetail the slice contents with the fresh instructions still being fetched.

In-order architectures rarely actually stall the processor when they encounter a cache miss ("stall on load"). Instead, they they allow independent instructions to proceed in the pipeline in parallel with the cache fill, but only up to the first instruction which uses the missing memory value, at which point they must stall ("stall on use").

In iCFP (in-order Continual Flow Pipelining) [22], this slice-out mechanism requires the much the same speculative-execution support an OOO does (ROB or equivalent result-buffering queue, load-store alias detection / avoidance), but allows the processor to make independent progress by not stalling on the first use of the result. Instead, iCFP poisons the result of the load and adds it to the slice buffer, and any later instruction that uses a poisoned value is sliced / poisoned in turn. When the miss returns, instructions are re-issued from the slice buffer in parallel to fresh instructions from the front-end (in the usual super-scalar fashion), potentially exposing new misses and MLP.

Since we assume the same speculation support in our experimental machine, iCFP makes an excellent compliment, and allows us to effectively extract another schedule in the shadow of a cache miss. Our implementation slices out the poisoned portions of each issue group into a FIFO, and always rallies to the head of that queue when any miss returns. Rallied issue groups are allowed to merge with fresh groups if they fit together, though no dependence checking is required for this merge due to the use of poison bits. See section 4.1 for its performance impact.

4. Results

In an effort to understand the roles that speculation and dynamism play in contributing to OOO performance, we first attempt to measure their relative contributions by measuring what fraction of OOO performance can be achieved through the OOO's speculation support without its ability to schedule dynamically (Section 4.1). We then move on to understand in what ways does dynamism contribute the remaining performance and to what degree do mechanisms that provide dynamism in the specific circumstances where it most benefits us can substitute for the general dynamism mechanism that is OOO execution (Section 4.2).

4.1 Speculation vs. Dynamism

To understand what fraction of baseline performance is contributed by OOO execution, we begin by comparing the performance of the baseline OOO to a comparably provisioned in-order superscalar, with both machines executing the traditionally compiled code. The

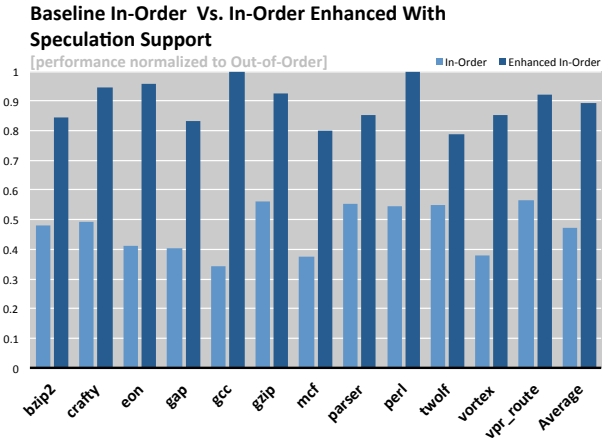

Figure 6. Performance Comparison of the Baseline In-Order and the In-Order Enhanced With Support for Software Speculation

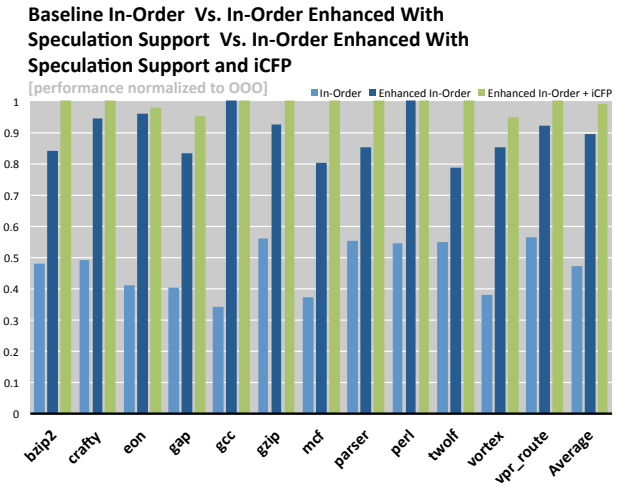

Figure 7. Performance Comparison of the Baseline OOO and the In-Order Enhanced With Support for Software Speculation and iCFP

in-order performance (normalized to the OOO) is shown in the left-most bar of Figure 6. We find the in-order machine to achieve, on average, 47% of the performance of the OOO, which is consistent with previous results [22]. This same work suggests that coupling a comparably provisioned in-order with an advanced run-ahead mechanism, iCFP is only sufficient to reach 57% of the OOO's performance, for the integer programs currently under investigation.

The pre-scheduled code exploits the speculation support of an out-of-order without benefiting from dynamism. Running the pre-scheduled code on the same in-order processor model results in substantially higher performance. As shown in the right bars of Figure 6, the pre-scheduled code achieves 89% of the OOO performance compared to the baseline in-order's 47%. From this data, it appears that it is OOO's ability to generate aggressive schedules, and not its ability to vary these schedules based on dynamic events, which is responsible for most of its performance advantage. Specifically, speculation provides roughly 79% of the gap between the in-order and OOO baselines.

4.2 Understanding Dynamism

By comparing the execution schedules produced by the OOO simulator with that of the pre-scheduled in-order, we observed two situations that seem to contribute the bulk of the performance discrepancy: re-scheduling in the locus of a cache miss and in the locus of a branch misprediction. In an attempt to quantify their relative contributions as well as to understand to what degree these types of dynamism could be captured by mechanisms that are potentially simpler than a general out-of-order mechanism, we undertook two further experiments.

First, we enhanced our simulator to implement a simple implementation of in-order Continual Flow Pipelining (CFP) [22], as described in Section 3.5. The dynamism that iCFP provides is very specialized and at a relatively course granularity. If there are no cache misses, iCFP has no impact on the execution. Upon a cache miss, the program is split into two streams, a miss-dependent stream and a miss-independent stream. Each of these streams retains its existing schedule, but these schedules can slip with respect to each other.

In spite of the somewhat simplistic nature of this dynamism, it seems to provide exactly what the pre-scheduled in-order execution requires. With iCFP, the pre-scheduled code can, on average, achieve 99% of the performance of the OOO, as is shown in Figure 7. We find that this makes a lot of sense. The pre-scheduled code already encodes the code re-ordering that is required for common case good performance, so the fact that iCFP does nothing in the absence of a cache miss is not a problem.

However, even in this configuration, the performance of `eon`, `gap` and `vortex` still lags behind the OOO by 2%, 4.9% and 5.2% respectively. We attribute this performance deficit to the in-order's inferior "recovery after branch mispredict" behavior described in detail in Section 2.2 with visualizations of such behavior excerpted from `eon`, `gap` and `vortex`. As an expedient, we found that simply executing the next 20 instructions after a branch mispredict on the dynamic scheduler enabled these benchmarks to match the OOO's performance; no more than 4% of the total instruction count is directed towards the dynamic scheduler in this manner. We further observed that the schedules discovered by the dynamic scheduler were remarkably stable suggesting a simple software only solution to this performance issue: branch-after-mispredict in which the front-end is redirected to an alternate version of the correct path schedule which is optimized for the instruction readiness that occurs during the mispredict recovery interval.

We also note (but do not show) that the combination of iCFP and speculative scheduling support actually enables the enhanced in-order to surpass the performance of the OOO in some cases. This is due to the fact that while waiting for misses to return, the Execute stage in the enhanced in-order only stalls when the ROB is full. In contrast, the OOO is compelled to stall when the Instruction Queue is full; practical designs generally feature Instruction Queues that have a fraction (typically less than half) of the number of entries contained in the ROB [24].

5. Discussion

The performance of our simulated machine with these harvested schedules suggests that the dynamic flexibility of a full-blown OOO processor is (expensive) over-kill and shows us some of the architectural features a more statically-scheduled processor needs to achieve similar performance.

5.1 Benefit of Dynamism

The primary advantage of a dynamic scheduler is that it can schedule for the dynamically-exposed critical path, without having to store and retrieve those schedules. The OOO gets many schedules

from a single set of instructions, while our in-order requires explicit schedules to be provided for every cache-miss and branch-mispredict that it schedules after. While it seems likely that not all of these schedules contribute significantly to performance, the front-end which can support this combinatorial explosion of schedules may be infeasible. We expect that profile-guided JIT compilation could identify the critical misses and mispredicts, and even provide reasonable code-growth, but these does not eliminate our need for low-latency access to a second (or third, or fourth) schedule to execute in the shadow of these unlikely events. The trade off of dynamic scheduling is thus between bringing several schedules to the execution core, or generating them on-site.

5.2 What mechanisms do inorders need to generate OOO-like schedules?

Given the above, there are two pieces of the in-order architecture that are missing.

The first is a scheme for decorating the instruction stream with secondary schedules, to be used in the event of unexpected long-latency events. In the case of a cache miss, this is akin to a "branch-on-cache-miss" where control is revectored based on the hit signal from the data cache similar to the mechanisms proposed in [23] and [1]. This obviously has little advantage if the schedule cannot see the use before the miss returns in addition to the more general problem of generating good miss-handling routines. The other mechanism is a "branch-on-branch-mispredict", where the schedule injected in the front end of the machine might be different, based on whether that path was predicted correctly, or it was the result of a re-steer.

The second piece of the puzzle is a mechanism for doing software alias speculation. The OOO uses a load-store queue to great effect, dynamically teasing out dependencies through memory, which allows for dynamic schedules that are aggressive where a compiler must be conservative. A non-faulting memory instruction, combined with a "branch-on-alias" would allow for the common-case execution to reap the benefits of not-always-safe load hoisting without sacrificing correctness. An OOO may also employ alias-prediction hardware, but the overhead of software scheduling adaptation in the presence of often-aliasing instructions may be prohibitively expensive, so this prediction mechanism may still be desirable (akin to branch prediction) in the in-order design.

6. Related Work

There is an extensive and venerable body of literature focusing on the fundamental advantages and disadvantages of dynamically scheduled hardware particularly in the context of attempting to match the OOO's performance with less complex hardware [4, 7, 30, 38, 44, 50]. While insightful and prescient in many respects especially regarding the scalability of dynamically scheduled hardware, the small memory footprints and low branch complexity of the programs used for comparison purposes in these studies make it challenging for the reader to discern what contribution dynamism, independent of speculation support mechanisms, makes to overall performance. Follow-on studies examined the static vs. dynamic scheduling issue in the context of the emerging memory wall and power-constrained but performance-intensive platforms [15, 39]. These works make strong arguments for a middle-ground between statically and dynamically scheduled hardware particularly due to perceived impending circuit limitations on OOO cycle times as well as concerns over object compatibility with the proposed simpler hardware.

Industry R&D [12] and academia [13] were exploring this middle-ground between statically and dynamically scheduled hardware, ("quasi-dynamic scheduling" in the parlance of rePLay[35]) while contemporaneous commercial processors at two extremes were the Itanium [40] and Transmeta processors [27]. Both the Itanium and Transmeta designs provide extensive hardware support for software speculation; data and control speculation are facilitated by associative structures and non-trapping instructions that the software can manipulate and query. Recovery from misspeculation differs in that the Transmeta design features automatic HW rollback [9] while Itanium exposes explicit rollback and recovery to software. Subsequently, Itanium designs have refined/expanded their HW structures to permit more control and data speculation [31] and the most recently announced version has greater HW support for dynamism [48] in a manner explored by [6]. We also note the encouraging trend of expanded HW support for software speculation, specifically the HW transactional memory systems in IBM's BlueGene/Q and Intel's forthcoming Haswell architecture [25].

Some form of transactional memory or HW atomicity is typically required [33] for quasi-dynamic scheduling but even a limited form combined with specialized functional units can facilitate performance comparable to a narrow-window OOO [8]. Though we do not address it in this work, quasi-dynamic and static scheduling all suffer from the region boundary issue that does not afflict the OOO; the OOO's ability to rotely eliminate control-flow and object code barriers to hoisting instructions is difficult to match both in theory [45, 49] and practice [17]. Matching the OOO's performance requires, at a minimum, the immensely challenging task of devising good predictors that function on large regions [17] while the HW support required to "region slip" is non-trivial.

Coping with memory operations remains problematic for both static and quasi-dynamically scheduled machines as they are typically compelled to stall on the use of an unready register. Simply finding sufficient instructions to cover load-to-use latency on an in-order can be a challenge and can introduce complications as seen in the mispredict recovery examples above. Some proposed solutions include dispatching instructions to latency-specific queues based on latency information encoded into the instructions at compile time [18] to more dataflow oriented ISAs and associated HW [32] though all come at the cost of significantly higher HW complexity and instruction encoding bloat.

Solutions for handling variable latency loads in general purpose in-orders range from the extremely aggressive hardware prefetchers, software prefetching support and runahead execution of the Power6 [5] to more modest compiler-inserted prefetch threads which use an idle SMT context [26]. An excellent overview of runahead techniques that occupy the middle ground between these extremes is provided by [3]. We chose iCFP [22] due to its recentness, its focus on in-order processors and the merits of its "rally" and "advance" modes which collectively enable it to outperform most rival techniques in the literature. We found iCFPs treatment of implementation issues equally compelling, though probably the greatest challenge facing runahead techniques is validation/verification. Finally, the microarchitectural substrate required to implement iCFP (multiple, checkpointed register files, advanced store queue, etc) could also provide HW support for software data and control speculation if these mechanisms were exposed at the ISA level.

Lastly, the observation that the OOO continually generates high quality schedules has motivated several research efforts to capture/re-use these schedules or to allow a compiler/runtime to indicate when dynamic schedules should be generated. The major motivation in this regard has been energy savings, [47, 49], generally at the cost of performance degradation though off-the-critical-path HW rescheduling of ROB traces in [34] yields a performance speedup over the baseline OOO. It must be noted that these approaches invariably execute a significant percentage of their instructions via the dynamic scheduler (upwards of 70%, 45% and 20% for [34, 47, 49] respectively) rather than executing the captured OOO traces on the in-order HW . Our own experiments

suggest that even executing only 5% of `bzip2`'s instructions via the dynamic scheduler enhanced performance by 18% relative to executing all instructions on the in-order HW. Finally, a novel and inspirational use of captured OOO schedules was demonstrated in [11] in which programs destined to run on an in-order architecture were first passed through a simulated OOO-version of the architecture, their traces captured and then appropriately patched up (in the form of compensation code for control/data misspeculation) to run directly on the in-order architecture.

7. Conclusions

Out-of-Order processors generally attain higher performance on control intensive integer code than in-order designs, including those with hardware support for software speculation. Conventional wisdom for this disparity in performance is the OOO's ability to avoid head-of-line blocking and exploit far-flung memory-level parallelism; collectively we call this ability "dynamism"; the major enabler for dynamism is the OOO's general-purpose, well-provisioned speculation support mechanisms e.g. large ROB, renamer, Load/Store Queue etc which can rollback speculative state at lower cost than even the most advanced in-order designs. We demonstrated that nearly 88% of the speedup attained by the OOO over an in-order design can be attributed to these speculation support mechanisms alone. Of the remaining performance differential, a complexity-effective run-ahead mechanism, iCFP, was sufficient to close the gap to the OOO in most cases. The final 1-2% performance gap was found to be in the in-order's inability to match the OOO's recovery after a branch mispredict; we proposed and qualified a simple mechanism for addressing this deficiency.

8. Acknowledgments

We are deeply indebted to Naveen Neelakantam and Sara Baghsorkhi, both of Intel Corp., for reviewing an early draft of this work. We also thank the anonymous reviewers for their insightful comments and suggestions. Daniel S. McFarlin was supported by an National Defense Science and Engineering Graduate Fellowship.

References

[1] B. A. Babaian, S. K. Okunev, and V. Y. Volkonsky. Critical path optimization–unload hard extended scalar block. USPTO 6584611, 2001.

[2] R. D. Barnes, J. W. Sias, E. M. Nystrom, S. J. Patel, J. N. Navarro, and W.-m. W. Hwu. Beating in-order stalls with "flea-flicker" two-pass pipelining. *IEEE Trans. Comput.*, 55(1):18–33, Jan. 2006.

[3] A. T. Brian Kreskamp, Pablo Montesinos. Enhancing mlp: Runahead execution and related techniques. IACOMA Technical Report 512, 2005.

[4] M. Butler and Y. Patt. An investigation of the performance of various dynamic scheduling techniques. In *Proceedings of the 25th annual international symposium on Microarchitecture*, MICRO 25, pages 1–9, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[5] H. W. Cain and P. Nagpurkar. Runahead execution vs. conventional data prefetching in the ibm power6 microprocessor. In *ISPASS*, pages 203–212, 2010.

[6] L. Carter, W. Chuang, and B. Calder. An epic processor with pending functional units. In H. Zima, K. Joe, M. Sato, Y. Seo, and M. Shimasaki, editors, *High Performance Computing*, volume 2327 of *Lecture Notes in Computer Science*, pages 445–448. Springer Berlin / Heidelberg, 2006.

[7] P. P. Chang, W. Y. Chen, S. A. Mahlke, and W.-m. W. Hwu. Comparing static and dynamic code scheduling for multiple-instruction-issue processors. In *Proceedings of the 24th annual international symposium on Microarchitecture*, MICRO 24, pages 25–33, New York, NY, USA, 1991. ACM.

[8] A. Deb, J. M. Codina, and A. González. Softhv: a hw/sw co-designed processor with horizontal and vertical fusion. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 1:1–1:10, New York, NY, USA, 2011. ACM.

[9] J. C. Dehnert et al. The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-life Challenges. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 15–24, 2003.

[10] J. Doweck. Inside Intel Core Microarchitecture and Smart Memory Access. Intel Whitepaper, 2006.

[11] M. Dupré, N. Darch, and O. Teman. VHC: Quickly Building an Optimizer for Complex Embedded Architectures. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 53–64, 2004.

[12] K. Ebcioglu and E. R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 26–37, June 1997.

[13] B. Fahs et al. Performance Characterization of a Hardware Framework for Dynamic Optimization. In *Proceedings of the 34th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2001.

[14] B. A. Fields, S. Rubin, and R. Bodik. Focusing processor policies via Critical-Path prediction. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 74–85, July 2001.

[15] J. Fritts and W. Wolf. Evaluation of static and dynamic scheduling for media processors. In *Proceedings of the 2nd Workshop on Media Processors and DSPs*, Micro '00, 2000.

[16] J. S. Gardner. Mips aptiv cores hit the mark. *Microprocessor Report*, May 2012.

[17] M. Gebhart, B. A. Maher, K. E. Coons, J. Diamond, P. Gratz, M. Marino, N. Ranganathan, B. Robatmili, A. Smith, J. Burrill, S. W. Keckler, D. Burger, and K. S. McKinley. An evaluation of the trips computer system. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 1–12, New York, NY, USA, 2009. ACM.

[18] J. P. Grossman. Cheap out-of-order execution using delayed issue. In *Proceedings of the International Conference of Computer Design*, CD 2000, pages 549 – 551, 2000.

[19] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic processing in cell's multicore architecture. *IEEE Micro*, 26(2):10–24, Mar. 2006.

[20] T. R. Halfhill. Netlogic doubles up xlp. *Microprocessor Report*, April 2011.

[21] M. Heffernan. *Data-Dependency Graph Transformations for Instruction Scheduling*. PhD thesis, Massachusetts Institute of Technology, 2007.

[22] A. Hilton, S. Nagarakatte, and A. Roth. icfp: Tolerating all-level cache misses in in-order processors. *IEEE Micro*, 30(1):12–19, Jan. 2010.

[23] M. Horowitz, M. Martonosi, T. C. Mowry, and M. D. Smith. Informing memory operations: Providing memory performance feedback in modern processors. In *In Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 260–270, 1996.

[24] Intel. Intel 64 and ia-32 architectures optimization reference manual. Intel Technical Manual, 2012.

[25] Intel. Intel architecture instruction set extensions programming reference. Intel Technical Manual, 2012.

[26] D. Kim and D. Yeung. Design and evaluation of compiler algorithms for pre-execution. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ASPLOS-X, pages 159–170, New York, NY, USA, 2002. ACM.

[27] A. Klaiber. The Technology Behind Crusoe Processors. Transmeta Whitepaper, Jan. 2000.

[28] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden.

IBM POWER6 microarchitecture. *IBM J. Res. Dev.*, 51:639–662, November 2007.

[29] D. J. Lilja. Reducing the branch penalty in pipelined processors. *Computer*, 21(7):47–55, July 1988.

[30] C. E. Love and H. F. Jordan. An investigation of static versus dynamic scheduling. In *Proceedings of the 17th annual international symposium on Computer Architecture*, ISCA '90, pages 192–201, New York, NY, USA, 1990. ACM.

[31] C. McNairy and D. Soltis. Itanium 2 processor microarchitecture. *IEEE Micro*, 23(2):44–55, Mar. 2003.

[32] R. Nagarajan, S. K. Kushwaha, D. Burger, K. S. McKinley, C. Lin, and S. W. Keckler. Static placement, dynamic issue (spdi) scheduling for edge architectures. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 74–84, Washington, DC, USA, 2004. IEEE Computer Society.

[33] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles. Hardware atomicity for reliable software speculation. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 174–185, 2007.

[34] O. Palomar, T. Juan, and J. J. Navarro. Reusing cached schedules in an out-of-order processor with in-order issue logic. In *Proceedings of the 2009 IEEE international conference on Computer design*, ICCD'09, pages 246–253, Piscataway, NJ, USA, 2009. IEEE Press.

[35] S. J. Patel and S. S. Lumetta. rePLay: A Hardware Framework for Dynamic Optimization. *IEEE Transactions on Computers*, 50(6):590–608, 2001.

[36] S. J. Patel, T. Tung, S. Bose, and M. M. Crum. Increasing the size of atomic instruction blocks using control flow assertions. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 33, pages 303–313, New York, NY, USA, 2000. ACM.

[37] N. Ranganathan, R. Nagarajan, D. Jimnez, D. Burger, S. W. Keckler, and C. Lin. Combining hyperblocks and exit prediction to increase front-end bandwidth and performance. Technical report, 2002.

[38] B. R. Rau. Dynamically scheduled vliw processors. In *Proceedings of the 26th annual international symposium on Microarchitecture*, MICRO 26, pages 80–92, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[39] K. W. Rudd and M. J. Flynn. Instruction-level parallel processors-dynamic and static scheduling tradeoffs. In *Proceedings of the 2nd AIZU International Symposium on Parallel Algorithms / Architecture Synthesis*, PAS '97, pages 74–, Washington, DC, USA, 1997. IEEE Computer Society.

[40] H. Sharangpani and K. Arora. Itanium processor microarchitecture. *IEEE Micro*, 20(5):24–43, Sept. 2000.

[41] J. L. Shin, H. Park, H. Li, A. Smith, Y. Choi, H. Sathianathan, S. Dash, S. Turullols, S. Kim, R. Masleid, G. Konstadinidis, R. T. Golla, M. J. Doherty, G. Grohoski, and C. McAllister. The next-generation 64b sparc core in a t4 soc processor. In *ISSCC*, pages 60–62, 2012.

[42] G. Shobaki. *Optimal Global Instruction Scheduling Using Enumeration*. PhD thesis, University of California Davis, 2006.

[43] G. Shobaki, K. Wilken, and M. Heffernan. Optimal trace scheduling using enumeration. *ACM Trans. Archit. Code Optim.*, 5(4):19:1–19:32, Mar. 2009.

[44] M. D. Smith, M. Horowitz, and M. S. Lam. Efficient superscalar performance through boosting. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 248–259, 1992.

[45] F. Spadini, B. Fahs, S. Patel, and S. S. Lumetta. Improving quasi-dynamic schedules through region slip. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '03, pages 149–158, Washington, DC, USA, 2003. IEEE Computer Society.

[46] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual flow pipelines. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XI, pages 107–119, New York, NY, USA, 2004. ACM.

[47] E. Talpes and D. Marculescu. Execution cache-based microarchitecture power-efficient superscalar processors. *IEEE Trans. Very Large Scale Integr. Syst.*, 13(1):14–26, Jan. 2005.

[48] S. Undy. Poulson: An 8 core 32nm next generation intel itanium processor, 2011.

[49] M. G. Valluri, L. K. John, and K. S. McKinley. Low-power, low-complexity instruction issue using compiler assistance. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 209–218, New York, NY, USA, 2005. ACM.

[50] D. W. Wall. Limits of instruction-level parallelism. *SIGARCH Comput. Archit. News*, 19(2):176–188, Apr. 1991.

[51] M. T. Yourst and K. Ghose. Incremental commit groups for non-atomic trace processing. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 67–80, Washington, DC, USA, 2005. IEEE Computer Society.