Accurate Critical Path Analysis via Random Trace Construction

Pierre Salverda Charles Tucker Craig Zilles

Department of Computer Science
University of Illinois at Urbana-Champaign
{salverda,ctucker2,zilles}@uiuc.edu

ABSTRACT

We present a new approach to performing program analysis through profile-guided random generation of instruction traces. Using hardware support available in commercial processors, we profile the behavior of individual instructions. Then, in conjunction with the program binary, we use that information to fabricate short (1,000-instruction) traces by randomly evaluating branches in proportion to their profiled behavior.

We demonstrate our technique in the context of critical path analysis, showing it can achieve the same accuracy as a hardware critical path predictor, but with lower hardware requirements. Key to achieving this accuracy is correctly identifying memory dependences in the fabricated trace, for which purpose we use a form of abstract interpretation to identify aliasing store-load pairs without explicitly profiling them. We also demonstrate that our approach is very tolerant of the quality of profile information available.

Categories and Subject Descriptors: I.6.3 [**Computing Methodologies**]: Simulation and Modeling–*Applications*; C.0 [**Computer Systems Organization**]: General–*Hardware/Software Interfaces, Modeling of Computer Architecture*

General Terms: Measurement, Performance

Keywords: Instruction Criticality, Profiling, Trace Fabrication

1. INTRODUCTION

Critical path information has been demonstrated to be effective for reasoning about a machine's performance bottlenecks [7, 8, 19], as well as for optimizing complexity-effective implementations of aggressive microarchitectures [6, 8, 12, 18, 21, 22]. This has lead to a number of proposals for engineering critical path detectors and predictors directly into the hardware. However, such schemes consume a non-trivial amount of silicon (*e.g.* 14KB in one proposal [8]), require tight integration with the execution core, and involve a significant amount of switching activity and hence power consumption. Our thesis is that these costs are avoidable: an online critical path infrastructure is, to a large extent, superfluous because instruction criticality can be computed just as effectively offline.

We present a scheme that uses basic profile information, plus a memory image of a program's code segment, to fabricate *random* traces that are representative of that program's execution. Those traces, which are annotated with details on microarchitectural events (*e.g.* branch mispredictions, cache misses), and which include memory alias information, are processed by a simple timing model to identify the static instructions that frequently appear on the critical path. The resulting criticality information proves to be just as effective as that derived from a sophisticated online infrastructure. For example, Figure 1 shows that, in the context of clustered microarchitectures, which we use as a vehicle for evaluating the efficacy of our scheme, criticality information derived from our fabricated traces provides 93% of the performance benefits achieved by dynamic, hardware-based critical path predictors.

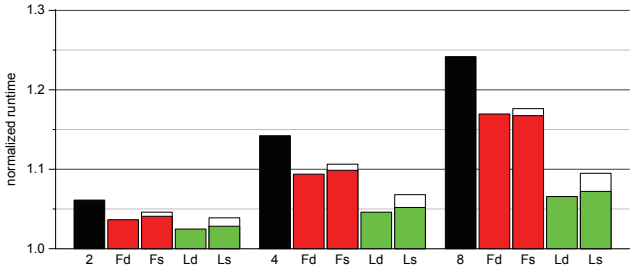


Figure 1. The potential of static criticality. The graph shows the runtime of 2-, 4- and 8-cluster machines relative to a resource-equivalent 8-wide out-of-order superscalar. (More details of our simulation infrastructure, and these performance results, are provided in Section 5.) Bars labeled '2', '4' and '8' show the performance of a standard dependence-based steering policy, which ignores instruction criticality. The 'Fd' and 'Ld' bars show the impact of taking criticality into account, first using the scheme proposed by Fields *et al.* [8], and then using the likelihood of criticality (LoC) policies developed by Salverda and Zilles [18]. Both use *dynamic* critical path detection and prediction logic in hardware. The 'Fs' and 'Ls' bars show *static* versions of those two schemes, where each instruction is tagged with a fixed criticality value derived from our trace fabrication scheme. The unshaded portion of those bars shows the performance implications of not fabricating memory dependences in our traces.

A high-level view of our proposed infrastructure appears in Figure 2. Broadly, the scheme comprises two components: *trace fabrication* and *trace analysis*. The former, which we describe in detail in Section 3, is the more important of the two, since it is the quality of the traces that we fabricate that ultimately determines the utility of the criticality information we collect. Those traces are constructed so as to reflect the same flow of control and the same dataflow relationships typically observed in a real execution, both of which are important for correctly distinguishing the instructions that tend to be critical from those that don't. Control flow is fabricated by taking a profile-guided random walk through the executable. Section 3.2 demonstrates the efficacy of this technique.

We use a novel technique for synthesizing memory dependences. While register dependences can be derived directly from a trace

CGO'08, April 5–10, 2008, Boston, Massachusetts, USA.

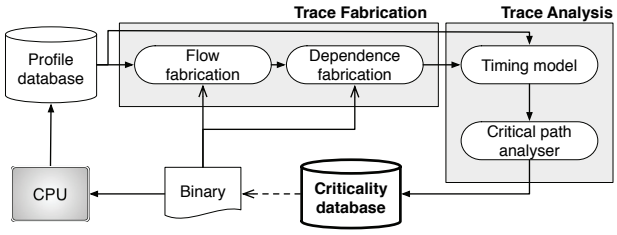


Figure 2. Our offline critical path infrastructure.

once control flow has been fabricated, memory-carried dependences — specifically, bypassing store-load pairs — are not obvious from the program itself, nor are they easily profiled. They are, however, important for accurately computing criticality. The unshaded portions of the 'Fs' and 'Ls' bars in Figure 1 show that removing synthesized memory dependences from our traces reduces the efficacy of the resulting criticality information to the point that the static schemes now achieve less than 75% of the benefit of their dynamic counterparts. In Section 3.3, we show how we use an abstract interpretation technique to compute symbolic values for load and store addresses and, thereby, correctly identify over 95% of all the aliasing store-load pairs.

The trace analysis component, which we describe in Section 4, uses a simple timing model to annotate each instruction in the fabricated trace with timestamps reflecting their progress through a simulated pipeline, one that models, at a high level, the same constraints imposed by the host CPU. It also uses profile data to associate microarchitectural events like branch mispredictions and cache misses with specific instructions in the trace, in so doing exposing those instructions, together with their backward dataflow slices, as critical path candidates. The final step — critical path analysis — simply delineates a critical path through the timing-annotated trace, for which purpose we use the dependence graph technique from Fields *et al.* [8] (described in Section 2.1). The resulting *criticality database* records the tendency of each static instruction to reside on the critical path.

Though our focus in this paper is on mechanisms for generating the criticality database offline, it is difficult to evaluate the quality of that database in the abstract: its efficacy is determined, ultimately, by the performance that can be derived from it. As we noted above, we use for this purpose two criticality-based instruction steering and scheduling schemes for clustered microarchitectures. Section 5 presents that study.

In addition to validating our trace fabrication approach, our results implicitly demonstrate two properties of criticality that are fundamental to the viability of any such offline scheme. First, because the fabricated criticality is computed without any knowledge of correlated events in the trace (*e.g.* branch B is taken if and only if branch A is taken), we can conclude that such information is largely unnecessary for computing criticality. Second, there is little need for adaptivity in criticality prediction: our trace fabrication approach computes a single *static* criticality value for each instruction, yet it achieves performance that is very close to that achieved by a hardware predictor. In this regard, we further find that an instruction's criticality is largely independent of program input (data presented in Section 5), permitting previous program inputs to be used to predict instruction criticality.

In summary, we make the following contributions in this paper.

1. We propose profile-guided random construction of traces for program analysis.
2. We demonstrate the use of abstract interpretation to identify memory aliases.
3. We demonstrate the efficacy of these techniques in the context of critical path prediction.
4. We show that instruction-level criticality tends to be stable and insensitive to program input.

Though we focus here specifically on critical path prediction, we believe random trace construction has other applications. It represents a "sweet spot" between the extremes of detailed microarchitectural simulation and abstract program analysis. With respect to the former, it can be performed more efficiently and requires neither building a complete simulator nor having to ensure that a representative sample of the program is simulated. With respect to the latter, it naturally incorporates both microarchitectural and dynamic dataflow information into the analysis, which program analysis does not (easily) capture.

2. BACKGROUND AND MOTIVATION

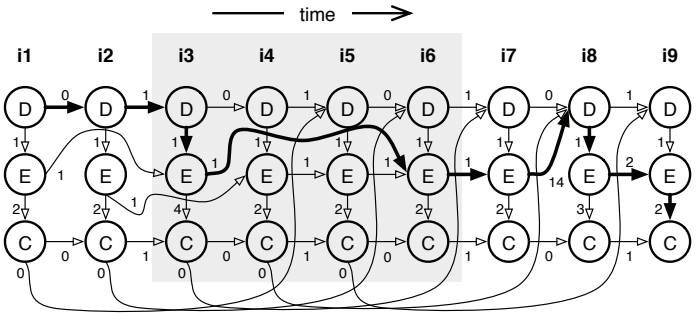
Informally speaking, the critical path through a program is a sequence of instructions whose aggregate execution time constitutes the runtime of that program. Our focus in this paper is on mechanisms for delineating that sequence and, hence, for isolating the critical instructions in a program. The ability to do so offers numerous benefits. From a performance point of view, knowing which instructions are critical permits compiler and hardware optimizations to be *focused* so that they target specifically those instructions that actually affect performance. For example, compilers for statically-scheduled machines give priority to critical instructions when they perform their scheduling pass [5]. Analogous techniques have been proposed for dynamically-scheduled machines. In particular, numerous studies have shown that criticality can be used to good effect in guiding instruction steering and dynamic scheduling policies in clustered superscalars [8, 18, 21].

In addition to those performance-enhancing benefits, criticality can play an important role in achieving more effective balancing of the various trade-offs between a design's performance potential and its complexity and power constraints. For example, a designer might reduce the clock frequency of some parts of the execution pipeline (to reduce power), and use instruction criticality to ensure that performance-critical instructions are preferentially allotted issue slots at the faster functional units [6, 20]. In so doing, non-critical instructions will be penalized, but since these typically exhibit some degree of *slack*, overall performance need not suffer.

More recent work has also demonstrated the utility of criticality as an analytical tool for diagnosing performance bottlenecks in novel microarchitectures [19]. The ability to delineate the critical path enables precise and quantitative attribution of the observed runtime to just those aspects of performance that are actually responsible. The more traditional approach of using aggregate metrics (like cache miss rate, for example) often fail to expose the most important factors responsible for performance and, more importantly, do not offer insight into the relative contribution of each factor to the observed runtime.

2.1 Defining criticality

Exactly what constitutes the critical path through a program depends very much on the context in which criticality is being used. In a statically-scheduled machine, for example, the critical path is induced by the schedule generated by the compiler, which is, in turn, dictated by the longest dataflow chain in each scheduling region. By contrast, in dynamically-scheduled machines — the focus of this paper — criticality is a function of a program's dynamic



D-D	in-order fetch
D-E	dispatch before execute
E-C	execute before commit
C-C	in-order commit
E-E	dataflow dependence
C-D	finite ROB capacity
E-D	branch misprediction

Figure 3. Critical path model. The dependence graph on the left models a sequence of 9 consecutive instructions (*i1* through *i9*) in an executing program. The table on the right describes the microarchitectural and dataflow constraints captured by that graph. In this example, the underlying machine has a 4-entry ROB (shaded region), so C–D edges, which capture the effects of finite window size, connect every 4th instruction. Instruction *i7*, a mispredicted branch, induces an E–D edge to instruction *i8* to reflect the constraint that correct path instructions cannot be dispatched into the window until a misprediction is resolved. The critical path through this code sequence is highlighted with the thicker dependence edges. All edges are labeled with their latencies.

dataflow patterns *and* their interaction with the underlying microarchitecture. Whereas the critical path in a statically-scheduled context is constant throughout execution, it can frequently change in a dynamically-scheduled context: when no microarchitectural events occur, the longest dataflow chain will be critical, but cache misses and branch mispredictions can elongate otherwise short dataflow chains, making them critical. In general, execution in a dynamic machine comprises a number of potentially-critical paths, their interplay being determined by events at runtime.

Though other studies have acknowledged the complexity of such interactions, Fields *et al.* were the first to tackle them directly and to characterize them precisely [8]. Our work focuses on their notion of criticality. Figure 3 depicts the *dependence graph* model upon which they build their critical path concepts. Very briefly, the model defines three main events for each instruction as it moves through the machine's pipeline: entry into the out-of-order window (dispatch), execution at a functional unit (execute), and exit from the out-of-order window (commit). Each dynamic instruction in a program execution thus contributes three nodes to the dependence graph; edges between those nodes capture the various dependences that constrain the order and time at which the corresponding events can occur. With this dependence graph in hand, the critical path is easy to define precisely: it is the *longest path* from the first 'D' node to the last 'C' node in the graph.

2.2 Delineating the critical path

The dependence graph model is inherently postmortem in nature: it can only identify the critical path for an execution that has already occurred. To render it a more practical tool, Fields *et al.* used it as the basis for a token-based critical path detector that could be incorporated into the processor pipeline [8]. This detects dynamic instructions whose execution (*i.e.* the E-node in Figure 3) resides on the critical path. The resulting outcomes are used to train a critical path predictor, a PC-indexed table of saturating counters. The predictor is analogous to a counter-based branch predictor, in the sense that it aggregates the past behavior of each static instruction into a binary — in this case, a critical/not critical — decision for subsequent dynamic instances. Fields *et al.* evaluate their scheme by using critical path predictions to guide instruction steering and scheduling decisions in clustered superscalar machines. More recent work by Salverda and Zilles extended those results by showing that mapping dynamic behavior onto a binary notion of criticality is too coarse-grained in a number of cases [18]. They propose instead a *likelihood of criticality* (LoC) predictor that records the fraction of an instruction's instances that are critical, and show likewise how LoC can further improve the performance of clustered machines.

These online schemes are appealing in two respects. First, they are flexible enough to adapt to changes in program behavior, and can therefore improve the efficacy of whatever pipeline optimizations use their criticality predictions. Second, since those optimizations will frequently change the critical path (*e.g.* value prediction can shorten it), an online scheme provides the ability to respond to the very optimizations it drives. In spite of these benefits, however, an online infrastructure comes at a cost. The token-based predictor, for example, requires a multi-ported 1.5KB array for detecting critical instructions, plus modifications to the execution core to record very detailed information on the relative timing of various events in an instruction's lifetime; and the predictor itself requires a table of 16K 6-bit saturating counters. Viewed in this light, an online critical path infrastructure incurs a significant cost — enough, perhaps, to mitigate the benefits of the criticality-aware optimizations it enables. Our principal claim in this work is that sophisticated online infrastructures are superfluous. Accurate criticality information can be derived offline using a profile-guided synthetic trace fabrication scheme, the details of which we present in Sections 3 and 4.

2.3 Related work in synthetic traces

Given that profiled program behavior plays a central role in our infrastructure, our approach shares a number of similarities with previous work that uses profile data to reproduce statistically similar behavior in an offline context. For example, statistical modeling [9, 14] and statistical simulation [15, 16] both make use of probabilistic models to reproduce, in aggregate, the kind of behavior observed by the real program. While we likewise exploit probabilistic properties of program behavior, we differ fundamentally from those prior studies in that an aggregate result is not sufficient for our purposes. Specifically, we are not interested in reproducing an IPC figure offline, but rather in examining dynamic sequences of instructions that are representative of the real execution.

Our work is therefore most closely related to the *shotgun profiling* technique proposed by Fields *et al.* [7]. That scheme also uses profiled behavior to feed an offline analysis of instruction criticality. However, whereas we profile only very basic dynamic events (details in the next section), shotgun profiling snapshots very low-level events pertaining to an instruction's progress through the pipeline. Though simpler than a full online critical path detection

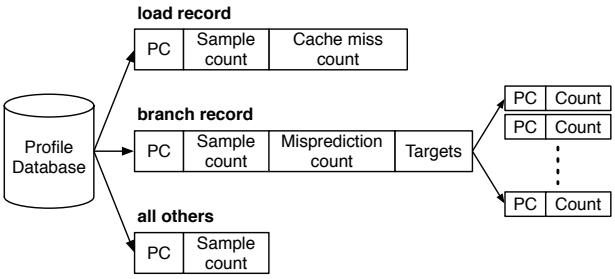


Figure 4. Profile data. We assume 3 types of records in our profile database: one for load instructions, one for branches, and one for all others. All record a sample count for the corresponding static PC. Loads and branches also maintain a record of microarchitectural events (cache miss or branch misprediction) for that static instruction. We also maintain a histogram of branch targets for each branch.

infrastructure, shotgun profiling still constitutes a substantial addition to the pipeline. We obviate the need for that hardware by relying much more on the original program binary to ensure that our traces embed the same dataflow patterns seen in a real execution.

3. TRACE FABRICATION

In this section, we describe our trace fabrication infrastructure in detail. We noted earlier that it is important that we produce detailed instruction traces that are representative of those observed in a real execution. By this, we mean that the traces we generate should embed the same dataflow patterns that are typically present in a real execution and, more importantly, that they facilitate mapping the important dynamic dataflow patterns back onto the static instructions that induce them. This requirement is conveniently viewed as comprising two parts. The first, and the more important of the two, is *control flow fabrication*. Section 3.2 describes how we use a profile-guided "random walk" through the program binary to fabricate the synthetic flow of control. The second component is *data dependence fabrication*, which we describe in Section 3.3. Our focus there is on fabricating dependences through memory, since dependences through registers are naturally induced by the control flow we fabricate. Since profile data seeds this whole process, we begin in Section 3.1 with a brief discussion of the profiling infrastructure we rely on.

3.1 Profiling infrastructure

Our trace fabrication infrastructure is driven by a database of profiled program behavior (recall Figure 2). Since that database contains information on microarchitectural events like cache misses and branch mispredictions, we rely on hardware support for profiling. Our requirements in this regard are modest, however. We need to precisely attribute events to the instructions that cause them, but this is a feature already available in many commercial microprocessors [4, 11, 13], as well as proposed for *AMD*'s recently announced *Lightweight Profiling* [1]. We will therefore not describe here exactly how the profile information is collected; more important is the specific information we rely on.

Figure 4 shows the program properties we profile. The bulk of our profile database is just a histogram of sampled PCs. In addition to that, we record counts of cache misses and branch mispredictions for each static load and branch that we sample. These are used by the trace analysis part of our infrastructure (Section 4). We also maintain a histogram of branch targets for each branch instruction that we sample. This is used by the control flow fabrication logic described in the next sub-section.

Algorithm 1 Control flow fabrication

```
 1: callStack ← ∅
 2: currPC ← GenerateRandomSeed()
 3:
 4: for i ← 0 to TRACELET_LENGTH do
 5:    currInst ← ReadFromBinary( currPC )
 6:    nextPC ← currPC + 4
 7:
 8:    if currInst is a call instruction then
 9:       callStack.push( nextPC )
10:       nextPC ← GuessTarget( currPC )
11:    else if currInst is a return instruction then
12:       nextPC ← ( callStack.empty() ? GuessTarget(currPC) : callStack.pop() )
13:    else if currInst is a branch or jump instruction then
14:       nextPC ← GuessTarget( currPC )
15:    end if
16:
17:    AddToTrace( currInst )
18:    currPC ← nextPC
19: end for
```

We impose two requirements on our profile data. First, like any profile-based technique, we require that a profile cover the program's code footprint with enough samples that we can readily identify the hot portions of the code. We characterize the sensitivity of our results to the quantity of profile information in Section 5. In principle, the requisite sample count can be achieved either by profiling over a short period of time with a high sampling rate or over a longer period of time with a low sampling rate. Second, we require that sampling be unbiased, in the sense that the number of samples we take for a given static PC is proportional to the relative frequency at which that instruction executes. This ensures that our trace fabrication logic will produce traces whose code coverage corresponds to that in a real execution.

3.2 Control flow fabrication

The fidelity of the criticality information we collect is determined, first and foremost, by the extent to which our traces faithfully reproduce the control flow observed in the real program. This is because control flow induces dataflow, and dataflow has a first-order effect on the critical path. Put another way, if we get the control flow right, we immediately get most of the dataflow right, as register dataflow is derived directly from control flow.

3.2.1 Flow fabrication logic

Algorithm 1 shows the main steps in our flow fabrication logic. We operate at a granularity of *tracelets* — short sequences of instructions seeded by randomly picking a starting PC from the profile database. The trace analysis component of our infrastructure, which we describe in Section 4, repeatedly consumes these tracelets in order to delineate the critical path.

Random seeding. The `GenerateRandomSeed` function used by Algorithm 1 (line 2) is responsible for picking the PC that will seed the tracelet generation. It operates by randomly selecting a record from the profile database, weighting the choice of an instruction by its sample count. In this way, we tend to seed traces in hot code regions. The ensuing control flow fabrication, described next, ensures that tracelets tend also to remain in hot regions.

Random walking. The for-loop in Algorithm 1 (lines 4–19) constitutes a profile-guided random walk through the program binary. We use the program binary to guide sequential control flow (lines 5 and 6) and the profile database to randomly pick the successors of control flow instructions (lines 8–15). For function `return` instructions (line 12), we maintain a simple call stack from which we

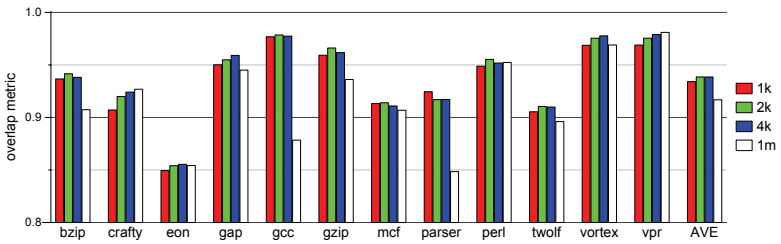


Figure 5. Path profile overlap. Moving left to right in each group, bars '1k', '2k' and '4k' plot the value of $M_f(P_r, P_f)$ for tracelet lengths of 1-, 2- and 4-thousand instructions; the '1m' bars correspond to 1-million instructions per tracelet. Each plotted value is computed by running trace fabrication for a total of 60-million instructions (60-thousand tracelets for bars labeled '1k', for example). In all cases, the profile databases driving the trace fabrication logic are collected from the same real program trace against which the fabricated traces are compared.

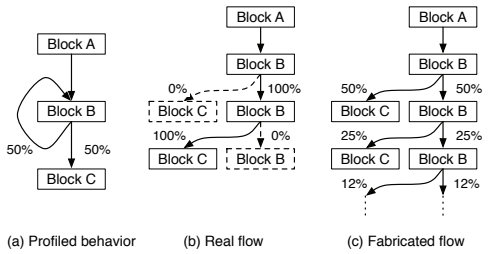


Figure 6. Correlated flow in `eon`. (a) The static control flow graph is distilled from hot method `ggSpectrum::Set`. Edges are annotated with their profiled frequencies. (b) Only one path through this code is ever observed at run time: exactly 2 iterations through the loop. (c) Our fabrication logic tends to explore all possible paths through the code.

pop target addresses; `call` instructions push their successor onto this stack (line 9). In this way, we establish semantically meaningful control flow across function call boundaries. The target addresses for `call` instructions (line 10), direct and indirect `jump` (unconditional) instructions, and conditional `branch` instructions (line 14), are all computed by the `GuessTarget` function.[1] Analogous to random seed generation, `GuessTarget` randomly picks a target PC from among all the profiled targets for the given control flow instruction, weighting the choice by those targets' sample counts.

3.2.2 Discussion

The above logic ensures that our tracelets are very likely to mimic real program control flow, repeatedly iterating through hot loops and tending to follow the most biased conditional paths within them. To evaluate the extent to which this is indeed happening, we draw on previous work in the area of *path profiling*. We use a path profile as a succinct measure of the control flow embedded in a given trace, and hence for comparing our fabricated traces to those from a real execution: the more closely the two match, the more accurately our traces reflect real execution. We follow the approach taken by Ball and Larus, who define a path as an acyclic sequence of intra-procedural basic blocks [3]. Unlike them, however, we do not use the so-called Wall weight-matching metric to compare two path profiles, preferring instead to use the *overlap metric* employed by Arnold and Ryder [2]. This is a more intuitive means for comparing two path profiles, and it is readily extended to also quantify the accuracy of dataflow dependence fabrication (Section 3.3).

Very briefly, the overlap metric is defined as follows. The flow along path p in path profile P, denoted $f(p, P)$, is the the total number of times path p occurs in profile P. The coverage of p in P, which is the fraction of total flow in P that can be attributed to p, is then given by $c(p, P) = f(p, P)/f(P)$, where $f(P) = \sum_{p \in P} f(p, P)$. The flow overlap metric, denoted M_f, compares real-trace path profile P_r to fabricated-trace profile P_f as follows: $M_f(P_r, P_f) = \sum_{p \in P_r} min\{c(p, P_r), c(p, P_f)\}$. This quantifies the extent to which the fabricated profile agrees with the real one on the fraction of total flow that is attributable to each of the real profile's constituent paths. Two identical path profiles will yield an overlap value of exactly 1. Any differences between the two will yield a value less than that: too little (or no) coverage of a given path by the fabricated trace will be captured by the min operation; too much coverage will be capped by the min, and will ultimately be penalized by the ensuing deficit of flow along other paths.

[1] Unconditional, direct (PC-relative) jumps are a special case. These have a statically-fixed target instruction that can be discovered from the binary.

Figure 5 plots the value of the overlap metric for the SPEC Integer benchmarks. That data is collected by first generating a real instruction trace (one derived from a program execution) of a given length. From this, we compute P_r, real-trace path profile. In addition, we generate a profile database for the real trace, and use that to fabricate a collection of tracelets whose aggregate length equals the size of the real trace; from this, we compute P_f. The data in Figure 5 shows the value of $M_f(P_r, P_f)$. Since we compare the fabricated trace to the same one from which its profile database is derived, these results constitute a form of self-training. We do so at this point to avoid clouding the results with artifacts of profiling specifics, but we will show later in the paper that removing these idealized assumptions has little impact on our results. The graph confirms that our fabricated traces are remarkably effective at reproducing the control flow observed in a real execution. On average, close to 95% of the real trace's flow is matched by the fabricated traces. This data is consistent with results published by Ball *et al.* [3], who found that a greedy strategy for deriving a path profile from an edge profile (*i.e.* a profile of branch targets, like ours), is able to very closely match the real path profile. This is testament to the highly-biased nature of most control flow.

The graph also shows that longer tracelets marginally improve the accuracy of the path profile, but only to a point. The '1m' bars, for example, show that very long tracelets generally yield a worse overlap value, and in some cases show marked degradations (`gcc` and `parser`). This is because, as tracelets grow in size, the total number of times we reseed the fabrication logic is reduced. Using fewer seeds increases the chance that our random walk ventures down a cold path and remains there too long. This can distort the extent to which our traces correctly reflect the relative time spent in each code region, as well as the likelihood that they properly cover all the executed regions. Our approach of repeatedly reseeding the flow fabrication is key to avoiding such problems. In this regard, very short tracelets are most appealing, but we must balance that requirement against the need for sufficiently long traces to capture all the dataflow and microarchitectural constraints that have bearing on the critical path. Tracelets shorter than the machine's ROB size, for example, will fail to capture the C–D edges in the dependence graph model. From our empirical evaluation of various alternatives for tracelet length, we find that 1,000 instructions strikes a good balance between the above factors, as well as minimizes the number of instructions we have to analyze before we obtain sufficiently accurate criticality profiles. We will therefore confine our analysis in the remainder of the paper to tracelets of 1,000 instructions each.

Figure 5 also shows that our flow fabrication is not uniformly successful across all benchmarks. The `eon` program, in particular,

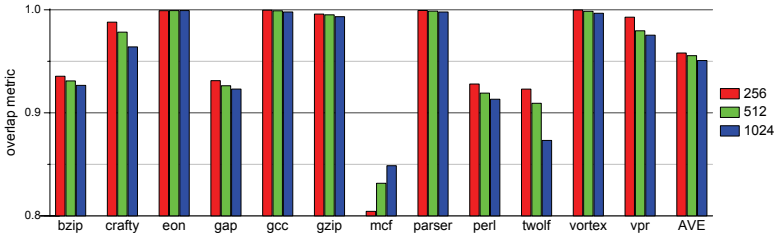


Figure 7. Load alias profile overlap. Each bar plots the value of the alias overlap metric for a real program trace relative to one whose memory addresses have been replaced with those generated by our symbolic execution technique. That is, we decorate a real trace with symbolic addresses, and compare the resulting alias profile with that of the original trace. Aliasing is measured in windows of 256, 512 and 1024 contiguous instructions, which we slide across traces comprising a total of 60-million instructions.

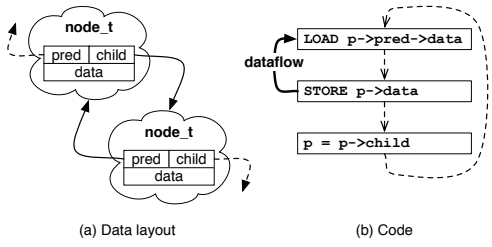


Figure 8. Correlated addresses in `mcf`. (a) The doubly-linked `node_t` data structure induces mutually dependent values in the `pred` and `child` fields. (b) Simplified version of an inner loop in hot method `refresh_potential`, which iterates over the linked structure, updating each node's fields based on the results of computation on its predecessor.

stands out. In that case, the less effective path coverage arises because of *correlated* control flow, which our fabrication logic does not take into account. Figure 6 shows why this is a problem in `eon`. Only a subset of all possible paths is observed during execution of this code, but our trace fabrication approach tends to enumerate all of them, thus watering down the path profile. These effects are comparatively rare, however, and, as we shall see, they tend to have little impact on the integrity of the criticality we extract from our fabricated traces.

3.3 Data dependence fabrication

From a critical path point of view, dataflow dependences through the register file are by far the most important. Because we fabricate traces based on actual instructions read from the binary (Algorithm 1, line 5), we faithfully model all intra-block register file dataflow. And because most of the control flow is representative of real execution, inter-block dataflow is mostly accurate too.

A more challenging problem is posed by dataflow through memory. As we saw in Figure 1, capturing the main store-to-load communication exhibited by the program is important for accurate identification of critical instructions. We therefore require a mechanism for fabricating aliasing behavior in our traces. To be clear, we make a distinction between *memory aliasing* and *memory bypassing*. A store and load alias if they touch the same address and there is no intervening store to that address; a store bypasses to a load if the pair alias *and* the store is still in the pipeline when the load issues. That is, bypassing is a pipeline-specific notion; aliasing is a property of the program. Our goal here is to generate synthetic *alias* information, not bypass information. The trace analysis logic (Section 4) uses that aliasing information, together with knowledge of pipeline parameters, to decide whether a given store should bypass to a given load.

In the absence of actual input data, we cannot, in general, know the exact memory addresses generated by a program. However, what matters most in determining memory aliasing behavior is not the addresses themselves, but simply whether two addresses are the same. The approach we take, therefore, is to perform an abstract execution of the tracelet using arbitrary values. Logically, we maintain a simple record of the machine's architected state, initializing every register and memory location with a random value, and then emulate the execution of each non-control flow instruction in the tracelet.[2] We record the "symbolic" addresses thus generated for each memory operation and later compare those values to identify aliasing store-load pairs.

[2] In practice, we only generate random values on demand, when a storage location that has not yet been written by the tracelet is read for the first time.

Proceeding in this way, computation within the trace is internally consistent; it is merely seeded with random input. Loads and stores that tend to alias in a real execution are therefore liable to also alias in our symbolic execution, albeit via meaningless addresses. Callee saved and restored registers, in particular, are easily detected: we seed the stack pointer with a random value, and subsequent saves and restores, which use fixed offsets relative to that random value, will correctly alias. A number of other aliasing patterns are likewise caught.

To get a feel for the efficacy of this technique, we develop the notion of an *alias profile* for an instruction trace. This is simply a histogram that associates a count with each static load-store pair that participated in a dynamic aliasing. If (st, ld) is one such pair, we define $c(st, ld, A)$ to be the fraction of all dynamic aliasing instances in alias profile A that are accounted for by that pair. This leads naturally to an aliasing overlap metric, the analog of the path profile metric M_f. Specifically, for real and fabricated alias profiles A_r and A_f, we define the alias overlap metric as follows: $M_a(A_r, A_f) = \sum_{(st,ld)\in A_r} min\{c(st, ld, A_r), c(st, ld, A_f)\}$.

Figure 7 plots the value of this metric for the alias profiles generated by our symbolic execution technique.[3] Overall, the data shows that the symbolic execution technique is remarkably effective at reproducing almost all the aliasing behavior observed in a real execution, with the overlap metric averaging about 95% across all the benchmarks. Moreover, the efficacy of the technique extends to large regions of contiguous instructions, rendering it useful for studying the effects of aliasing even in large-window machines.

Figure 7 exposes `mcf` as an outlier — the only benchmark to consistently score below 90%. Figure 8 shows why this is occurring. The problem here is analogous to the correlated control flow problem: certain values in a program's data structures are correlated, in the sense that program semantics establish some form of relationship between them. This becomes important for aliasing when those data values are pointers. In the `mcf` code, such a relationship exists between the `pred` and `child` fields of the nodes in a doubly-linked data structure.[4] With our symbolic execution technique, we seed values for these two pointer fields randomly, so we fail to establish any relationship between them. As Figure 8 shows, one of `mcf`'s hot loops performs an update to this linked structure,

[3] Actually, we compute a bi-directional overlap, since the metric, as defined, will falsely report good overlap if, for example, an alias profile deems all load-store pairs to always have aliased. To obviate such problems, Figure 7 plots the average of $M_a(A_r, A_f)$ and $M_a(A_f, A_r)$. The path profile overlap is not prone to this problem because total flow in the two profiles is fixed, so too much flow along one path will always be compensated for by less flow elsewhere; not so with alias profiles.

[4] Specifically, the two pointer fields are mutually referential: if `n` points to a non-leaf node, then it will always be the case that `n->child->pred == n`.

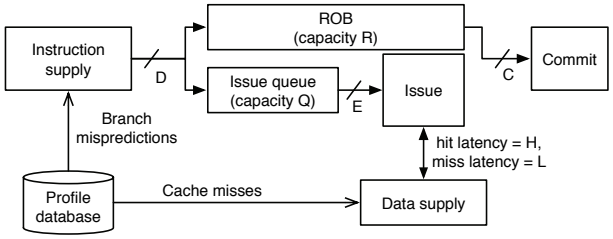


Figure 9. High-level view of the timing model.

doing so in such a way that updates to the fields of one node are used during the computation on the child node in the subsequent loop iteration. That is, there is frequent store-to-load bypassing across successive loop iterations. Fortunately, scenarios such as this are comparatively rare in the programs we studied, so they do not perturb our criticality analysis very much.

4. TRACE ANALYSIS

In this short section, we briefly describe trace analysis, the second component of our offline infrastructure. Recalling Figure 2, this comprises two main components: the timing model and the critical path analyser.

4.1 Timing model

The timing model consumes one tracelet at a time. Its task is to annotate each instruction with timing information to mimic the effects of that instruction's progress through a simulated pipeline. Those timestamps reflect the manner in which the fabricated dataflow is likely to interact with the low-level microarchitectural constraints imposed by the host CPU. It is that interaction which ultimately determines the critical path.

The principal requirement we impose on the timing model is that it faithfully model the microarchitecture of the host CPU. This does not mean we need a very detailed microarchitectural simulator, but rather that we need to capture the first-order effects that have the most influence on the critical path. Specifically, we need to model the machine constraints captured by the dependence graph shown earlier in Figure 3. This leads to the high-level model depicted in Figure 9, which represents a typical out-of-order superscalar machine. Our timing model is essentially a simple trace-driven simulator for an out-of-order machine. The main microarchitectural constraints we capture relate to peak instruction supply (fetch) rate, issue bandwidth (different rates for different instruction types), and peak commit bandwidth. We also model the effects of finite buffering capacity in the machine's out-of-order window. Instruction execution latencies are also modeled, but not necessarily true to the latencies imposed on the host CPU. Rather, the objective is to capture the effects of long-latency floating point operations and variable-latency load instructions, and to distinguish those from single-cycle integer operations.

Also shown in Figure 9 is the use of profile data to inject branch mispredictions and cache misses into the trace. These microarchitectural events have a strong influence over which static instructions tend to appear on the critical path. The timing model captures their effects by randomly selecting the dynamic instances of branch (load) instructions that will incur a misprediction (cache miss). This is done as per the profiled behavior of the corresponding static instructions. That is, branches that are frequently mispredicted in the real execution will tend to incur many simulated mispredictions in the timing model; likewise for cache-missing loads. We simulate a branch misprediction simply by stalling instruction supply for a fixed number of cycles. Simulated cache misses result in load instructions incurring a fixed number of cycles of additional latency.

Our approach of randomly deciding when branch mispredictions and cache misses occur will, of course, miss the effects of correlation among such events, as well as any tendency they might have to cluster in time. However, these effects are of secondary importance. What counts most is that branches that tend to mispredict, and loads that tend to miss in the cache, are frequently flagged as such in the simulated trace, and hence that they and their backward dataflow slices are liable to be exposed on the critical path.

In the empirical work we present in Section 5, where we evaluate our criticality infrastructure in the context of clustered superscalar machines, the CPU we use in our timing model is slightly different from the one depicted in Figure 9. In particular, its issue queue is partitioned, and dataflow between those partitions incurs a global communication penalty. We likewise need to model an instruction steering policy for distributing instructions among the clusters. While these components add a little more complexity to the timing model, they do not fundamentally change the basic infrastructure described above.

4.2 Critical path analysis

As we noted in Section 2, we use the dependence graph model from Fields *et al.* to delineate the critical path. We conduct the analysis on a per-tracelet basis, delineating the entire critical path for each before moving onto the next. Since we have each of the tracelets available in its entirety, we do not need to actually construct the whole dependence graph. Instead, we simply move backwards through each tracelet, using the embedded timestamps to (logically) follow the last-arriving edge at each of the implied nodes. Any instruction whose execution (*i.e.* its E-node) is reached by means of this backward traversal is deemed critical. For each static instruction thus identified, we increment two counters, one for the total number of dynamic instances seen, and another for the total number of critical instances seen; for all other instructions, we increment just the former counter.

Maintaining two counters for each static instruction in this manner permits us to derive both a binary criticality and a likelihood of criticality (LoC) profile from our analysis. In both cases, we use the ratio of critical to total instances to determine the final criticality value of each instruction. To obtain a binary value, we use a threshold ratio of 0.125 to make the not-critical/critical distinction. This corresponds roughly to the mechanism employed by the counter-based predictor proposed by Fields *et al.*, which has its counters incremented by 8 when training critical, and decremented by 1 when training not-critical. Though different threshold values may yield slightly better results, we find that the 0.125 serves adequately for our purposes. To derive an LoC value from the criticality profile, we simply take the ratio of critical to total instances. Without a significant loss of performance, LoC can be discretized to 8 values, representable in 3 bits [17].

Once multiple tracelets have been processed in this way, we write the entire criticality database to file. Obviously, the number of tracelets we process before doing so will have an effect on the accuracy of the criticality database we generate. In particular, processing more tracelets improves the extent to which we cover the complete binary. We present data in the next section to quantify this effect, and hence to gauge the amount of offline tracelet processing that is necessary before sufficiently accurate criticality databases are obtained.

Instruction supply	Perfect instruction cache. Aggressive tournament branch predictor.
Front-end	8-wide, 12 stages to dispatch.
Window	128-entry unified issue queue. 256-entry ROB.
Execute	Up to 8 instructions per clock, any mix of up to 8 integer, 8 floating point and 4 memory instructions (load or store). Latencies similar to the Alpha 21264 [10]. Perfect memory disambiguation.
Memory	32KB 4-way set associative L1 cache, 2 cycle access time. 8MB, 8-way set associative L2 cache, 12 cycle access time. DRAM latency 300 cycles.

Table 1. Monolithic baseline machine parameters.

5. EVALUATION

The previous sections have demonstrated that our fabricated traces closely match real program traces, both in terms of the control flow they embed and the dataflow they induce. Of course, the ultimate utility of our scheme is determined not by the integrity of the traces themselves, but by the quality of the criticality we derive from them. As a vehicle for evaluating this, we use the instruction steering and scheduling schemes for clustered microarchitectures that were briefly discussed in Section 2. We show in Section 5.2 that the criticality we generate from self-trained profile databases can be used to very good effect in static versions of those two schemes. In Section 5.3, we then proceed to remove some of our idealized assumptions about the profile databases we use. We show that the quality of the criticality we generate is not very sensitive to changes to the input of the program we sample, nor to the rate at which we collect samples. Finally, in Section 5.4, we quantify the amount of work that must be performed by our scheme before the criticality it generates is of sufficiently good quality. Before we present those experiments, we briefly describe our simulation infrastructure.

5.1 Empirical framework

We use a trace-driven timing simulator to evaluate three different clustered microarchitectures. Each is a different partitioning of a monolithic (*i.e.* not clustered) 8-wide out-of-order superscalar. Table 1 enumerates the salient parameters for that monolithic machine, which we use as a baseline throughout this section. The three clustered machine configurations divide its execution and issue queue resources equally among 2, 4 and 8 clusters.[5]

We simulate the SPEC 2000 Integer benchmark suite. We provide performance results from the reference input sets in all cases, and, in Section 5.3, also use the training input sets to generate our profile databases. All benchmark programs were compiled for the Alpha ISA using the DEC C Alpha compiler (V5.9-005), with peak optimization enabled, but with no profile feedback. We perform our simulations at 10 equally-spaced checkpoints across each benchmark's complete run. In each of those runs, we simulate 100-million instructions after warming up the memory system and branch predictor, giving a total of 1-billion instructions simulated per benchmark. When collecting our profiles, we produce one database for each benchmark, this being the aggregation of the behavior profiled during the 10 checkpoint runs.

We have implemented and evaluated two previous proposals for using criticality in clustered machines. The first is the work of Fields *et al.*, who used their token-based critical path detector and binary criticality predictor to: (1) steer instructions to the critical producer of their operands (where there is a choice); and (2) give precedence to critical instructions in the out-of-order issue queue at each cluster [8]. We will henceforth refer to this as the binary-criticality scheme. The second, which we simply call the LoC scheme, is the set of LoC-based steering and scheduling policies introduced by Salverda and Zilles [18]. They use the same token-based critical path detector as Fields *et al.*, but they instead train an LoC predictor. LoC predictions drive a collection of sophisticated instruction steering and scheduling policies, the details of which are not important here. Due to space constraints, we will present data in this section for the LoC scheme only, but we will refer to both in our discussion of results. (Average results for the binary scheme do appear in Figure 1, however.)

To incorporate our fabricated criticality databases into the above schemes, we developed for each of them a static version in which we simply tag static instructions with a *fixed* criticality value, either binary or LoC, as appropriate.

These schemes are themselves not the object of our study. They are merely vehicles for demonstrating the efficacy of our technique. That said, we want to emphasize that both constitute the state-of-the-art in clustered microarchitecture research. Indeed, obtaining the good IPC results published in the aforementioned studies is not easy, if at all possible, in the absence of criticality information.

5.2 Performance from self-training

So as to isolate the effects of extraneous variables, we begin with an evaluation of criticality derived from *idealized profiles*. By this we mean two things. First, the profile database used by our offline scheme is generated by sampling every instruction in the real execution. We thereby ensure complete code coverage. Second, we evaluate the quality of the resulting criticality by using it in simulation of the same traces we profiled — we self train. Section 5.3 explores the implications of removing these idealized assumptions.

Figure 10 shows how our static version of the LoC scheme compares to its dynamic counterpart. Results for the binary criticality scheme (not shown) follow exactly the same trends. In both cases, the average performance differences between the static and dynamic schemes are extremely small. In fact, the static versions are never more than 3% slower (`eon` on the 8-cluster configuration) and, on average, are less than 1% slower than the dynamic version. That `eon` stands out a little can be accounted for by the data in Figure 5, where we showed that correlated control flow in this benchmark causes some inaccuracies in our trace fabrication. Overall, however, the criticality we fabricate is very precisely reproducing the same average behavior achieved by an online critical path detector and predictor.

5.3 Sensitivity analysis

We now explore the extent to which the above good results depend on idealized profiling. There are two dimensions to this. The first is the input sets that drive the profiling runs. We show in Section 5.3.1 that criticality derived from profiles of the SPEC *training* runs, and then used to drive runs of the *reference* input sets, continues to deliver good performance. There are a few cases in which performance is not as good as the dynamic schemes, but this occurs because of a lack of code coverage — an absence of, not inaccuracies within, criticality data. The second dimension is the amount of profile data that must be collected. Section 5.3.2 quantifies the extent to which this affects the quality of fabricated criticality.

5.3.1 Program input

Criticality appears to be remarkably stable across different program input sets. We generated our profile databases from runs of the SPEC training input sets, and then fed those databases into our trace-fabrication infrastructure. The criticality databases thereby produced were then used to drive precisely the same set of refer-

[5] In the case of the 8-cluster machine, we round up the memory issue bandwidth to 8 load/store per cycle to avoid having to share 4 memory ports among 8 clusters.

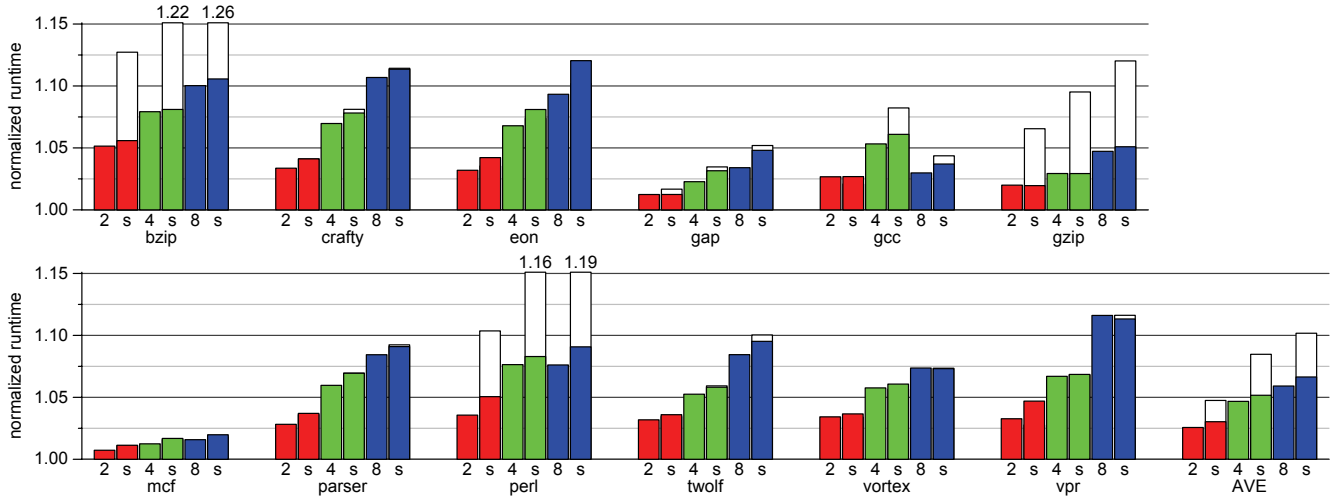
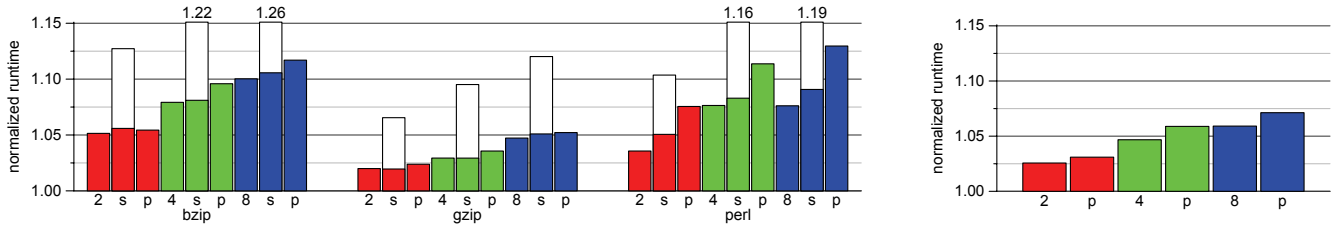


Figure 10. The potential of static criticality. The graph plots the runtime, relative to the monolithic baseline, of the LoC-based policies on three three different clustered machines. The SPEC reference input sets were used in all simulations. Bars labeled '2', '4' and '8' plot the performance of *dynamic* versions of the LoC-based policies for 2-, 4- and 8-cluster machines. Adjacent to each of those, bars labeled 's' plot the performance of the corresponding machine, driven now by *static* LoC data derived from our trace fabrication scheme. The shaded portions of those bars show the performance achieved when trace fabrication is driven by profiling of the reference inputs (*i.e.* self-training); the unshaded portion (not visible in most bars), by profiling of the training input sets.



(a) Patched criticality (b) New averages

Figure 11. Code coverage. The graph on the left shows that performance of the three worst-performing benchmarks can be drastically improved when the criticality database used by the static LoC scheme is supplemented with criticality derived from profiling of the reference input sets (only *missing* information is added; no existing criticality data is modified). The first two bars for each clustered machine repeat the data from Figure 10. Bars labeled 'p' show the performance with the patched criticality databases. The graph on the right shows how the overall performance averages change when just these three amended results are factored in.

ence input set runs reported on above. The results are shown by the unshaded portions of the bars labeled 's' in Figure 10. Performance is, in general, indistinguishable from the self-trained results: most benchmarks show little or no change. Only three — `bzip`, `gzip` and `perl` — stand out as suffering substantial performance losses. However, those losses can be ascribed to poor code coverage in the training input sets. To demonstrate this, we patched the training criticality database with some of the fabricated LoC derived from the reference inputs. That is, we supplement the criticality data by adding to it from the self-trained database, but we do so only for instructions for which we are entirely missing criticality information; we do not modify any existing data. Figure 11(a) shows that the three worst-performing benchmarks now all perform similarly to the rest. With these amendments to just those three benchmarks, the average performance bars from Figure 10 change to those shown in Figure 11(b) — very close, now, to the idealized results of Figure 10.

That code coverage appears to be the only cause for performance loss bodes well for an offline infrastructure in which there is continuous profiling of program execution. In such an environment, the profile databases can be repeatedly updated across multiple runs of an application, steadily improving the ability of the trace fabrication process to achieve complete code coverage.

5.3.2 *Quantity of samples*

A second factor that we idealized in Section 5.2 is profiling of every instruction in the real execution (we assumed a sampling rate of 100%). To explore our sensitivity to this assumption, we could change the sampling rate, but that, by itself, is not a very useful metric: the *timespan* over which sampling occurs can mitigate even a very low sampling rate. A more useful metric, then, is the total number of instruction samples needed before the fabricated traces become sufficiently representative of real execution. To measure this, we adopted an approach in which we distill small, fixed-size profile databases from the fully-sampled databases that we have been using up to this point. This is achieved by randomly picking profile records from the original databases, biasing the choices as per those records' sampled frequency. In a series of experiments, we pick a fixed number of samples — ranging from one thousand to one million — to generate a distilled database, which we then use to drive the trace fabrication logic. We then compare the performance obtainable from each of the resulting criticality databases.

Figure 12 shows how the performance of the static scheme improves as a function of the number of profile samples used to drive the trace fabrication infrastructure. Clearly, performance very rapidly converges on its final value, with all machine configurations not showing much improvement beyond 128-thousand samples.

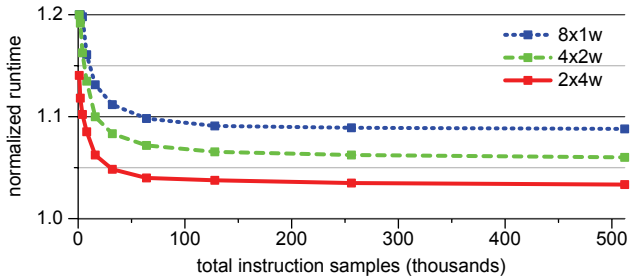


Figure 12. Performance as a function of sample count. Each line plots the harmonic mean (across all 12 SPEC Integer benchmarks) of the runtime of each of the 3 clustered machines relative to that of the monolithic machine. The x-axis shows the number of samples in the profile database that drove the trace fabrication scheme.

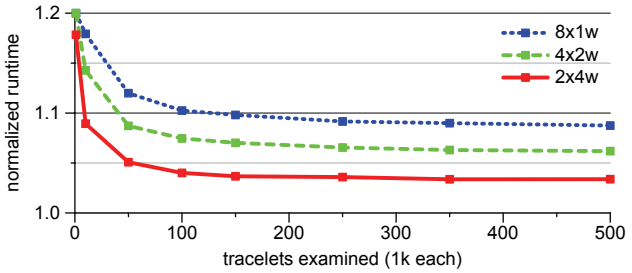


Figure 13. Performance as a function of tracelets generated. Like Figure 12, each line shows the harmonic mean of runtime on the 3 clustered machines relative to that on the baseline. In these experiments, we use a complete profile database, but run the trace fabrication logic for a fixed number of (1000-instruction) tracelets before emitting the criticality database.

5.4 IPC convergence

We conclude this section by showing that our offline logic tends to converge on its final criticality profiles very quickly. That is, we do not need to fabricate and analyze many tracelets before we obtain a sufficiently accurate criticality database. Figure 13 shows the performance of the three clustered machines relative to the monolithic baseline as a function of the number of tracelets examined before their criticality database is generated. Performance very quickly converges on its final value by the time we have processed on the order of 100-thousand instructions (*i.e.* 100 tracelets of length 1,000).

6. CONCLUSION

We have demonstrated that instruction criticality is rather stable, both within and across runs of a program, a property that renders it well-suited to offline analysis. We have also shown that criticality can be accurately computed using profile information obtainable from hardware already available in commercial processors. Our approach is built on two key observations. First, in assessing an instruction's criticality, we are really characterizing what fraction of a static instruction's dynamic instances are critical. Second, this fraction is stable to the extent that we can substitute representative for real execution traces in order to compute it. We proposed a profile-guided random trace construction method that exploits statistical properties of the program to generate representative instruction traces, plus an abstract execution method for discovering which store-load pairs in those traces are liable to alias. This approach is extremely effective, achieving upwards of 90% of the benefit of dynamic critical path predictors in hardware.

We believe that random trace construction is an interesting new analysis technique that can potentially serve a number of other applications. In particular, it offers a compelling design point between traditional trace-driven dynamic analysis and static compiler analysis. Furthermore, although the method is already modest in terms of its input data and processing requirements, we believe it could be further optimized to reduce the number of tracelets examined before results converge. For example, biasing trace seed selection to cover cold regions of the code, and compensating for this when aggregating the results, could potentially achieve better code coverage with a smaller number of tracelets.

7. REFERENCES

[1] Advanced Micro Devices, Incorporated. Lightweight profiling proposal. `http://developer.amd.com/assets/HardwareExtensionsforLightweightProfilingPublic20070720.pdf`, August 2007.

[2] M. Arnold and B. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 168–179, June 2001.

[3] T. Ball, P. Mataga, and M. Sagiv. Edge profiling versus path profiling: The showdown. In *Proceedings of the 25th Symposium on Principles of Programming Languages*, pages 134–148, January 1998.

[4] J. Dean, J. Hicks, C. Waldspurger, W. Weihl, and G. Chrysos. ProfileMe: Hardware support for instruction-level profiling on out-of-order processors. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 292–303, December 1997.

[5] J. Ellis. *Bulldog: A Compiler for VLIW Architectures*. PhD thesis, Department of Computer Science, Yale University, April 1986.

[6] B. Fields, R. Bodik, M. Hill, and C. Newburn. Slack: Maximizing performance under technological constraints. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 47–58, May 2002.

[7] B. Fields, R. Bodik, M. Hill, and C. Newburn. Using interaction cost for microarchitectural bottleneck analysis. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 228–242, December 2003.

[8] B. 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.

[9] N. Jouppi. The nonuniform distribution of instruction-level and machine parallelism and its effect on performance. *IEEE Transactions on Computers*, 38(12):1645–1658, December 1989.

[10] R. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March 1999.

[11] A. Mericas. The PowerPC performance monitor. In *Workshop on Hardware Performance Monitor Design and Functionality*, February 2005.

[12] G. Muthler, D. Crowe, S. Patel, and S. Lumetta. Instruction fetch deferral using static slack. In *Proceedings of the 35th International Symposium on Microarchitecture*, pages 51–61, November 2002.

[13] N. Nihdi. Performance monitoring on Pentium 4 processors. In *Workshop on Hardware Performance Monitor Design and Functionality*, February 2005.

[14] D. Noonburg and J. Shen. Theoretical modeling of superscalar processor performance. In *Proceedings of the 27th International Symposium on Microarchitecture*, pages 52–62, November 1994.

[15] S. Nussbaum and J. Smith. Modeling superscalar processors via statistical simulation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 15–24, September 2001.

[16] M. Oskin, F. Chong, and M. Farrens. HLS: Combining statistical and symbolic simulation to guide microprocessor designs. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 71–82, June 2000.

[17] N. Riley and C. Zilles. Probabilistic counter updates for predictor hysteresis and stratification. In *Proceedings of the 12th International Conference on High Performance Computer Architecture*, pages 110–120, February 2006.

[18] P. Salverda and C. Zilles. A criticality analysis of clustering in superscalar processors. In *Proceedings of the 38th International Symposium on Microarchitecture*, pages 55–66, November 2005.

[19] P. Salverda and C. Zilles. Dependence-based scheduling revisited: A tale of two baselines. In *6th Annual Workshop on Duplicating, Deconstructing, and Debunking*, June 2007.

[20] J. Seng, E. Tune, and D. Tullsen. Reducing power with dynamic critical path information. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 114–123, December 2001.

[21] E. Tune, D. Liang, D. Tullsen, and B. Calder. Dynamic prediction of critical path instructions. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pages 185–195, January 2001.

[22] E. Tune, D. Tullsen, and B. Calder. Quantifying instruction criticality. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 104–113, September 2002.