

Dynamic Dead-Instruction Detection and Elimination

J. Adam Butts and Guri Sohi
Computer Science Department
University of Wisconsin-Madison
{butts,sohi}@cs.wisc.edu

Abstract

We observe a non-negligible fraction—3 to 16% in our benchmarks—of *dynamically dead instructions*, dynamic instruction instances that generate unused results. The majority of these instructions arise from static instructions that also produce useful results. We find that compiler optimization (specifically instruction scheduling) creates a significant portion of these *partially dead* static instructions. We show that most of the dynamically dead instructions arise from a small set of static instructions that produce dead values most of the time.

We leverage this locality by proposing a dead instruction predictor and presenting a scheme to avoid the execution of predicted-dead instructions. Our predictor achieves an accuracy of 93% while identifying over 91% of the dead instructions using less than 5 KB of state. We achieve such high accuracies by leveraging future control flow information (i.e., branch predictions) to distinguish between useless and useful instances of the same static instruction.

We then present a mechanism to avoid the register allocation, instruction scheduling, and execution of predicted dead instructions. We measure reductions in resource utilization averaging over 5% and sometimes exceeding 10%, covering physical register management (allocation and freeing), register file read and write traffic, and data cache accesses. Performance improves by an average of 3.6% on an architecture exhibiting resource contention. Additionally, our scheme frees future compilers from the need to consider the costs of dead instructions, enabling more aggressive code motion and optimization. Simultaneously, it mitigates the need for good path profiling information in making inter-block code motion decisions.

1 Introduction

Many of the optimizations performed by compilers have costs associated with them, forcing the compiler to decide whether the benefits outweigh the costs in each particular instance. Heuristics or profile data can be used to guide these decisions, but these have problems. Heuristics are easily obtained but can fail when applied to unforeseen cases. Profile data, when available at all, can be an imprecise guide. The behavior of profiled code in certain phases might be vastly different from the final profile; different inputs might perturb the behavior from the profile as well.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ASPLOS X 10/02 San Jose, CA, USA
© 2002 ACM 1-58113-574-2/02/0010...\$5.00

An additional complication arises because the exact costs incurred by a particular optimization are not clear. Many compile-time optimizations create *partially dead instructions* [8]. These static instructions generate dead values only on certain paths through the program. Compilers often use estimated instruction count increase as a cost metric for partially dead code. In a dynamically scheduled processor core, however, the cost of a particular dead instruction is difficult to determine. A good compiler will schedule a partially dead instruction such that when its execution is unnecessary, it does not delay code that is useful. The dynamically dead instruction is far from free, however: its execution wastes processor resources, increasing utilization with no effect on the final computation. A direct consequence of this increased utilization is higher-than-necessary power consumption. Under resource contention, performance can suffer as well. Simultaneous multi-threaded processors, for example, depend on the availability of free resources to allow multiple threads to execute concurrently; increased resource utilization in these designs directly impacts the realized performance.

The myriad difficulties in cost estimation compel us to consider whether the compiler might optimize more aggressively if it could ignore costs completely and focus only on the projected benefits of a particular optimization. To answer this question first requires a means to reduce or eliminate the costs dynamically. To that end, we propose a mechanism that enables the hardware to eliminate dynamically dead instructions at run time. We conjecture that, given this advantage, future compilers could perform more aggressive optimizations, particularly those involving code motion, ignoring the costs of non-profitable and misguided optimizations.

We find that even without this safety net, today's optimizing compilers produce code with a non-negligible quantity of partially dead code. Most of this code arises during instruction scheduling when the compiler moves instructions across basic block boundaries. While the partially dead code does not often impact performance (since it is hidden by useful work and nothing needs to wait for its output), resource utilization is unnecessarily increased. Therefore, even without the performance benefit of more aggressive compiler optimizations, we can significantly lower the cost of executing existing codes.

In this paper, we present a detailed exploration of the characteristics of dynamically dead instructions. We show that a significant fraction of these instructions are introduced by the compiler during optimization. A relatively small set of partially dead static instructions produces most useless instances. Furthermore, useless instructions exhibit temporal locality—85% of useless instances arise from a static instruction that generated a useless instance on its prior execution. These properties suggest that the usefulness of an instruction instance is predictable.

We propose the use of a *dead instruction predictor* to enable hardware to detect dynamic instances that generate dead register values. We then present a scheme that acts on these predictions, avoiding the execution of these instructions, thereby reducing the utilization of several key resources. We observe notable reductions in resource utilization without impacting performance. Also,

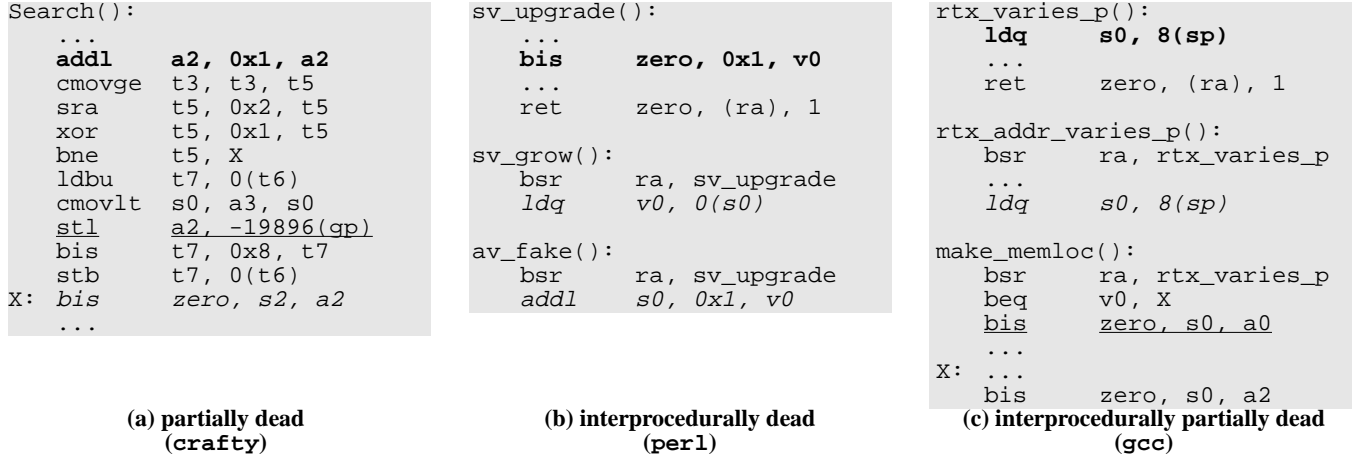


Figure 1. Examples of instructions that generate dead values (boldface). Uses of the value are underlined; kills are italicized. Ellipses indicate omitted code that contains no references to the values of interest. (a) The value in `a2` is partially dead: it is either overwritten by the `bis` instruction at X or used by the `stl` depending on the branch condition. (b) `sv_upgrade()`'s return value in `v0` is statically dead: *none* of the calling functions (e.g., the two shown here) use this value. (c) The function epilogue of `rtx_varies_p()` restores the callee-save register `s0`. This is required for some of the callers (e.g., `make_memloc()`), but in others (e.g., `rtx_addr_varies_p()`), `s0` is dead, rendering the restore useless.

we facilitate additional performance gains from future optimizing compilers that can apply code motion previously precluded by the associated costs, although the investigation of such compilers is not a part of this paper.

In the next section, we characterize dynamically dead instructions in detail, looking at their prevalence, relationship to compilation, and predictability. Section 3 presents and evaluates different strategies for the identification of dynamically useless instructions. In Section 4, we describe several methods of useless instruction elimination; these are evaluated in detail in Section 5. Then, we discuss related work in Section 6 before concluding in Section 7.

2 Dynamically Dead Instructions

In this section, we examine the types of dynamically dead instructions seen in typical integer codes. A dynamically dead instruction is an instance of a static instruction that generates a dead register value. For brevity, we frequently refer to a dead dynamic instruction as a *useless instruction*. First, we discuss the role of the compiler in the occurrence of useless instructions. Next, we examine the prevalence and characteristics of the useless instructions. Finally, we attempt to quantify the predictability of these instructions without regard to a specific predictor design.

Note that we focus exclusively on instructions that generate register values. While store instructions may be dead if the stored data is never again referenced (or if the store is silent [9]), the detection of such instances (particularly in a multiprocessor machine) is more complicated. Stores and branches are never considered useless because of the requirement that a register value be produced; for the same reason, nops and prefetches are also not classified as useless.

Our benchmarks are the integer codes of the SPEC2000 suite. All data collection was performed using the train inputs run to completion with the following exceptions: we ran only the routing phase of `vpr`, we used only the `kajiya` input for `eon`, and we ran only the `diffmail` program for `perl`.

2.1 Classifying dead instructions

To understand the existence of useless instructions, we begin by examining some specific examples of code that generates them. We subdivide the useless instructions into three broad classes: (1) dead or partially dead within a procedure, (2) dead or partially dead interprocedurally, and (3) dead or partially dead register restores. Figure 1 shows examples of these classes of instructions from optimized versions of our benchmarks. The partially dead instruction in Figure 1(a) increments a value used only on the fall-through path of a conditional branch. The corresponding source code performs the increment only on the necessary path. By virtue of hoisting the increment above the branch during optimization, the compiler was responsible for this partially dead instruction. The highlighted instructions in Figure 1(b) and Figure 1(c) require interprocedural analysis to properly classify them. With this analysis, elimination of the statically dead instruction in Figure 1(b) would be trivial; handling the partially dead restore of Figure 1(c) demands more complex (and costly) optimization algorithms such as procedure cloning [4].

As evident in the example code in Figure 1(a) (by no means an isolated example), the compiler has a major influence on the preponderance of useless instructions observed during the execution of a program. Figure 2 shows the percentage of useless dynamic instructions executed for representative benchmarks as a function of the compiler and optimization level. The two compilers we used were DEC C V5.9/C++ V6.1 and gcc/g++ version 3.0.3 (version 2.95.3 was used on the gcc benchmark due to a bug in the version 3.0.3 compiler). The different configurations represent different levels of optimization (`-O0` to `-O4` for the DEC compiler and `-O0` to `-O3` for gcc).[†]

It is immediately apparent that the fraction of useless instructions increases with the compiler optimization level. The absolute number of useless instructions (not shown) exhibits similar behav-

[†] Additional compiler options were `'-arch ev6 -non_shared -fast'` for the DEC compiler and `'-static -mcpu=ev6'` for gcc. These options select the EV6 architecture for scheduling purposes and result in a statically-linked binary. The `-fast` option for the DEC compiler turns on several other optimization flags, mostly enabling optimizations requiring assumptions (e.g., about aliasing, alignment, or required floating point accuracy).

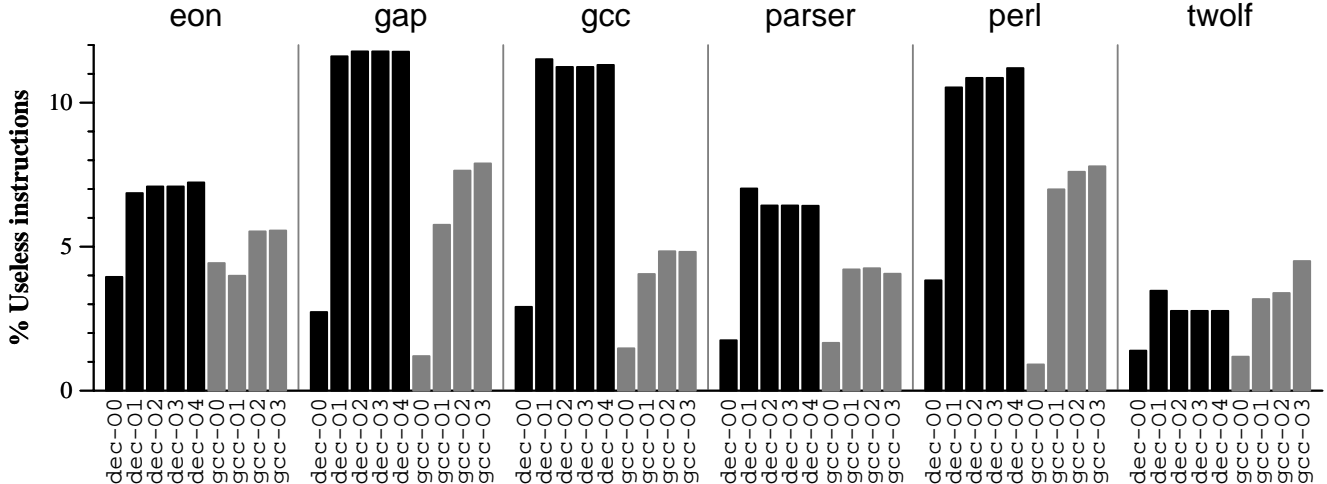


Figure 2. Effect of compiler optimizations on the fraction of useless instructions. Black bars represent DEC compiled benchmarks, while gray bars represent gcc compiled benchmarks. Of the omitted benchmarks, *bzip2* and *vpr* behaved similarly to *gap*; *vortex*, *gzip*, and *crafty*, to *parser*; and *mcf* to *perl*.

ior, increasing with greater optimization. However, even as the number of useless instructions increases, optimization is producing the desired overall effect: the number of cycles required to execute the benchmark (not shown in the figure) is decreased with more aggressive optimization.

The most significant increase in the number of useless instructions (and in performance) occurs as the lowest level of optimization is enabled. More interestingly, partially dead instructions within a single procedure (such as the example of Figure 1(a)) account almost entirely for the observed increase. We have isolated this effect and attributed it to the code scheduling phase of compilation. During code scheduling, the compilers move computations across branches to facilitate better performance on the expected target architecture [2, 3].

There are several reasons why a compiler might move instructions across basic block boundaries: (1) resource constraints, (2) long latency operations, and (3) reduced register pressure. All processors have a limited number of resources (e.g., execution units and cache ports). VLIW processors demand that the compiler be aware of these constraints to ensure that the resource requirements of instructions scheduled to execute simultaneously do not exceed the available resources. Although not a correctness issue for dynamically scheduled processors, resource-aware scheduling boosts performance since instructions that would oth-

erwise stall due to structural hazards can be replaced by useful work. Secondly, by initiating long latency operations (e.g., complex floating point instructions and loads expected to miss in the cache) as early as possible, there may be more work available to hide their latencies. Finally, many instructions take two register inputs and produce only one output. By moving such an operation up in the code, the live ranges of the two source registers may be reduced at the expense of a larger live range for the destination register. In some circumstances, this will allow register spills to be avoided in the block from which the instruction was moved.

2.2 Characteristics of useless instructions

Table 1 presents data on the useless instructions in our benchmarks (the DEC-compiled benchmarks at the `-O3` optimization level are used for the remainder of the paper). Dead values comprise 2.4% to 15.7% of the values produced during program execution, depending on the particular benchmark. The second numeric column of the table shows that in most benchmarks about 5% of the static instructions generate only useless values. On average, these statically dead instructions account for about 38% of the useless values in the program, although this varies from about 6% (*gzip*) to over 80% (*vortex*). About 11% of the static instructions (5.29% + 5.89%) generate at least one useless

Table 1. Useless Instruction Characteristics

Benchmark	Useless values (% dynamic)	Statically dead instructions (% static)	Statically dead instructions (% useless)	Partially dead instructions (% static)	Transitively useless values (% dynamic)
<i>bzip2</i>	2.44%	5.85%	72.55%	5.25%	0.88%
<i>crafty</i>	10.85%	3.98%	10.03%	7.96%	5.50%
<i>eon</i>	7.09%	4.97%	44.58%	3.19%	2.72%
<i>gap</i>	11.78%	6.22%	56.63%	6.36%	5.05%
<i>gcc</i>	11.24%	5.38%	21.42%	8.40%	4.01%
<i>gzip</i>	8.53%	5.08%	5.58%	5.85%	2.93%
<i>mcf</i>	15.68%	6.23%	6.32%	4.26%	1.99%
<i>parser</i>	6.43%	9.17%	14.17%	9.17%	2.29%
<i>perl</i>	10.86%	5.17%	58.08%	5.54%	3.29%
<i>twolf</i>	2.77%	3.25%	38.22%	4.50%	0.46%
<i>vortex</i>	14.79%	9.35%	80.15%	4.39%	3.90%
<i>vpr</i>	3.69%	4.18%	46.76%	5.77%	1.68%
Average	8.85%	5.29%	37.87%	5.89%	2.89%

value. For most of the benchmarks, the partially dead instructions account for the majority (62% on average) of unused values.

Transitively useless instructions—quantified in the rightmost column in Table 1—generate results only used by useless instructions or other transitively useless instructions. An example of commonly observed transitively useless code is the address generation code for a useless load. Counting transitively useless instructions increases the number of useless instructions by 33% on average (e.g., from 10.9% to $10.9\% + 5.5\% = 16.4\%$ for *crafty*). Previously published results [11] suggest that this number would be higher if values were tracked through memory, but we deliberately restrict ourselves to instructions generating register values since our eventual goal is the elimination of useless instructions. The external visibility of memory (to other processes, including the operating system) significantly complicates the elimination of stores.

Figure 3 breaks down the sources of useless instructions in more detail for a representative subset of benchmarks. Figure 3(a) shows the distribution by the propensity of each static instruction to generate a dead value. Thus, the intercept of each curve with the y-axis is equal to the number in the third numeric column of Table 1. The curves show that most useless instructions come from static instructions that are useless most of the time. For example, instructions that generate useful values during less than 20% of their executions comprise at least half of all useless instructions on all benchmarks; on average, 75% of useless instances are generated by instructions useless more than 75% of the time. Figure 3(b) shows the contribution of the instructions that generate the most useless instances regardless of how often those particular instructions are useless. In most cases, a small number (~300) of instructions accounts for 80% of all useless instances. Unsurprisingly, a greater number of static instructions contribute useless instances for those benchmarks with large code footprints (*gcc* and *crafty*). Examining the same data by percentage (instead of number) of static instructions shows similar trends.

Finally, we consider the breakdown of useless instructions by type. All but a fraction of a percent of useless instructions are ALU operations and loads. This is true even in those benchmarks where other types of instructions (e.g., integer multiplies or floating point operations) make up a non-negligible percentage of all dynamic instructions. About half of the benchmarks exhibit sig-

nificantly different ratios of ALU operations to loads among useless instructions than among all dynamic instructions. Most often, there is a higher fraction of loads among useless instructions, although the opposite is true for a few benchmarks. This phenomenon will affect the relative reductions obtained in the utilization of different resources when we eliminate useless instructions (Section 5.1).

2.3 Useless instruction predictability

Given a dynamic instruction that produces a dead value, our goal is to accurately identify that instruction as early as possible in the pipeline. The predictability depends on the working set of partially dead instructions and the behavior of each individual instruction. The former impacts a hypothetical predictor’s coverage (i.e., the availability of a prediction for a particular instruction). The larger the working set of dead and partially dead instructions, the more state required to obtain a certain level of coverage. The behavior of an individual instruction affects both predictor size and accuracy. When a particular static instruction generates a complex pattern of useless and useful instances, it will require more state to attain a reasonable prediction accuracy (if the pattern is predictable at all) than a static instruction that always generates dead values.

First, we consider the working set of useless instructions. Figure 3(b) indicates that several hundred static instructions are responsible for the majority of useless instances, but it gives little information on the working set of these instructions. Only a subset of those instructions is likely to be active in any given phase of the program’s execution. We characterize the working set of useless instructions by keeping a stack of static instructions and observing the depth at which a particular static instruction is located when that instruction is decoded. The decoded instruction is then moved to the top of the stack. Only instructions that have generated at least one useless instance modify the stack. The resulting coverage equals that of a perfect fully associative predictor with LRU replacement. Figure 4 shows the resulting working set data. We find that good coverage can be obtained using a relatively small predictor: with only two exceptions, the most recent 256 useless instructions cover 90% of subsequent useless instructions. Achieving comparable coverage with a less associative

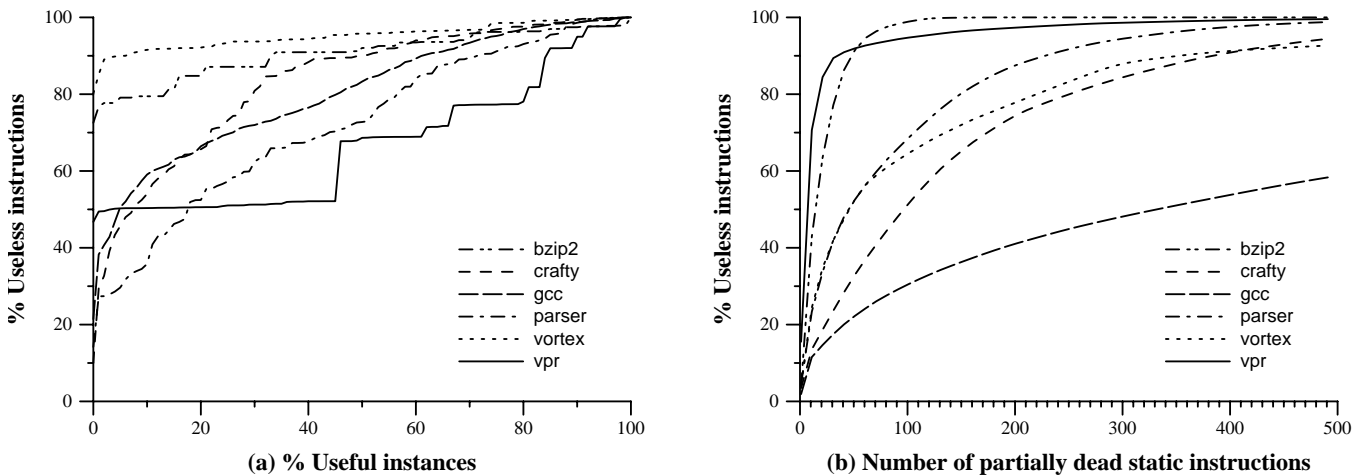


Figure 3. Cumulative useless instruction distributions. (a) The x-axis represents static instructions of a certain “usefulness.” The value of a curve at 20%, for example, shows the number of useless instructions generated by all static instructions that are useless over 80% of the time. Thus, points on the y-axis represent the contribution of statically dead instructions only. (b) The value N on the x-axis represents the top N static instructions (sorted by number of useless instances generated).

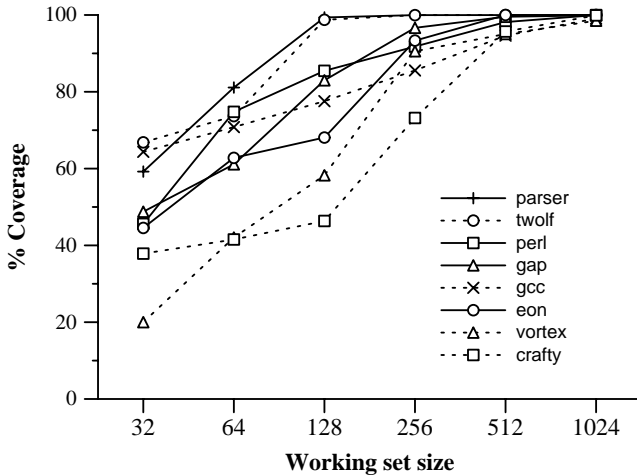


Figure 4. Working set of dead/partially dead instructions. This distribution shows the percentage of instructions for which a prediction would be available (coverage) given a fully-associative predictor of the given size with LRU replacement. A 32-entry predictor achieves 98% or better coverage on the omitted benchmarks.

predictor would naturally require more storage to overcome conflicts.

Determining whether a particular instance of a partially dead instruction is useless or not is the other part of predictability. The data in Table 1 and Figure 3(a) indicate that a significant portion of the useless instructions arise from partially dead code. Therefore, to be most effective at detecting these useless instances, we must be able to accurately differentiate useless and useful instances of a single static instruction. Fortunately, the “uselessness” of a particular instruction exhibits locality: on average, 85% of useless instructions follow another useless instance from the same static instruction. 99% of useless instances come from static instructions that generated useless instructions in one of the prior two executions. Given some additional information to decide which of the last outcomes to predict, we can hope to obtain high prediction accuracies.

3 Dead Instruction Prediction

Having seen that the usefulness of an instruction is amenable to prediction, we turn now to *how* it will be predicted. In this section, we are concerned solely with the evaluations of predictors themselves. Consideration of how the useless instructions will be exploited is the topic of the next section.

We employ two metrics in evaluating alternative predictor implementations. *Accuracy*—the first metric—quantifies the ability of the predictor to avoid potentially costly false positive predictions. It is simply the percentage of predictions made that are correct. We also have the notion of lost opportunity—one may achieve high accuracy simply by generating a few select predictions. Therefore, we also quantify the false negatives (i.e., the number of dead instructions that we did not identify) as well. We define a second metric called *coverage* that equals the percentage of all dead instructions that were predicted. In other words, higher coverage corresponds directly to lower missed opportunity.

First, we discuss a simple predictor similar in organization to a cache. Then we will augment the predictor to improve its performance, first by exploiting control flow information and then using a confidence mechanism. Finally, we will address the hardware cost of implementing a dead instruction predictor.

3.1 A simple dead instruction predictor

The simplest predictor design is a cache that remembers those instructions that were useless during their last execution. The cache is indexed by the PC of a value-producing instruction. The predictor may also be associative, allowing predictions to be made for multiple instructions mapping to the same entry. We evaluated a large number of predictors of varying size, associativity, and tag length.

Figure 5 shows the performance of a few representative configurations of the simple predictor (full timing simulation was used for all predictor evaluations; see Section 5 for details). To allow comparison across benchmarks with widely varying fractions of useless instructions (see Table 1), the bars are normalized to the number of useless instructions in each benchmark. Thus, the bars representing correctly predicted useless (gray) and unpredicted useless (white) instructions always total 100%. The black bars represent the number of useful instructions that were incorrectly

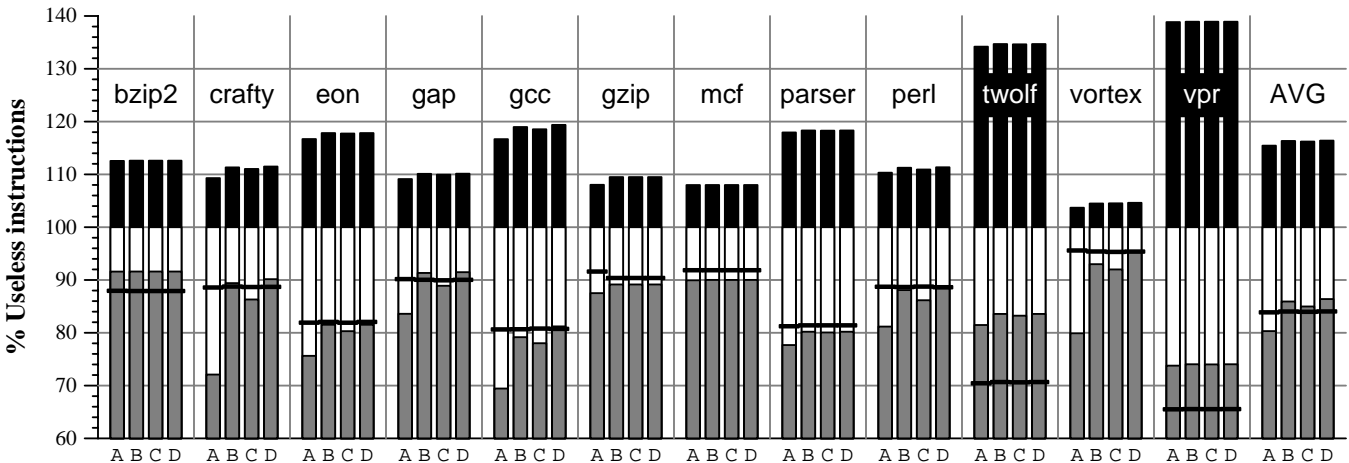


Figure 5. Performance of simple dead instruction predictor. The graph is normalized to the number of useless instructions in each benchmark. The gray bars represent correct predictions; the white, useless values not predicted; and the black, mispredictions. The horizontal lines indicate accuracy (correct predictions out of total predictions). The configurations are: (A) 512-entry, direct mapped; (B) 512-entry, fully-associative; (C) 2k-entry, direct-mapped; (D) 2k-entry, 4-way set associative. Note the offset of the y-axis.

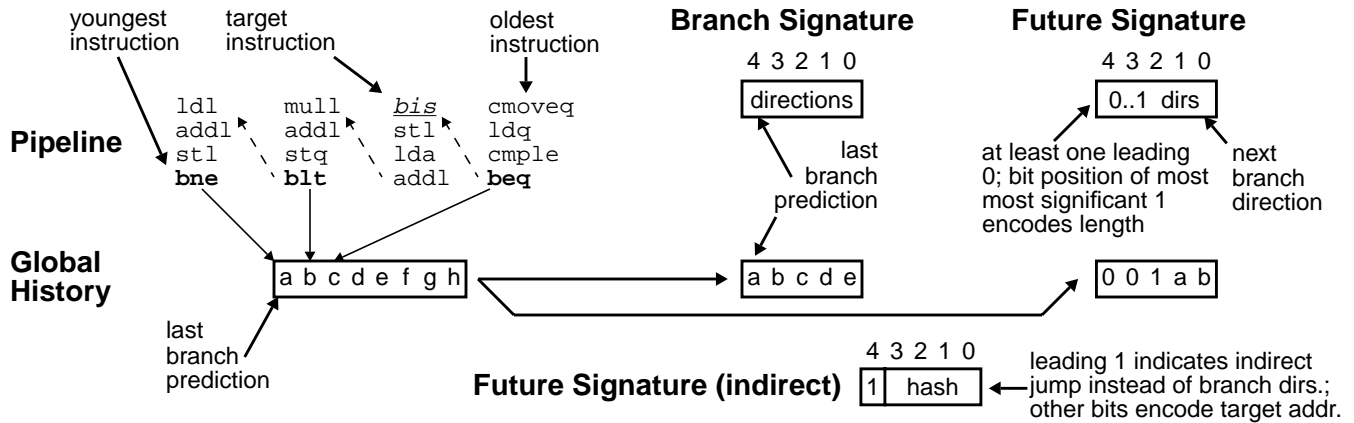


Figure 6. Control flow signatures. The generation of the two types of control flow signatures is shown for the instructions in a hypothetical pipeline. The branch signature consists of recent bits (5 in this case) from the global branch history. The future signature encodes the number and direction of branches occurring only *after* the target instruction (2 in this case). If the first control instruction after the target instruction is an indirect jump, the future signature encodes the hashed jump target (and the high order bit is set).

predicted as useless. Coverage may be read directly by the top of the gray bars. Horizontal black lines indicate accuracy; these are placed on the same scale to allow for easy comparison and correlation with coverage although they indicate a percentage of predictions rather than a percentage of useless instructions. The value is obtained by dividing the size of the gray bar (correct predictions) by the sum of the heights of the gray and black bars (total predictions).

As might be expected from the data presented in Section 2.3, we find that simple predictors perform reasonably well. On average, the base predictor organization (512-entry direct mapped) attains 80% coverage with 84% accuracy; most benchmarks exhibit at least 80% accuracy. The availability of a prediction in a simple (cache-like) predictor implies that the last instance of a particular instruction was useless. Therefore, the accuracy closely tracks the frequency of consecutive useless instances and is almost independent of predictor size and associativity.

While the accuracy of a simple predictor is nearly independent of predictor organization, coverage depends strongly on it. For the base predictor organization, the coverage is well correlated with the working set size of instructions (see Figure 4). The third and fourth configurations both quadruple the capacity of the base configuration: one gains more sets; the other, higher associativity. While both see increases in coverage, increasing the associativity helps more than increasing the number of sets by the same factor. The second configuration shows the benefit of associativity more clearly: a fully associative predictor performs nearly as well as the set-associative predictor four times larger.

As tag bits are the primary storage cost of a simple predictor, we also experimented with varying tag lengths. Unsurprisingly, we found (data not shown) that accuracy degrades as the tags become shorter. The accuracy penalty depends primarily on the amount of aliasing (due to the relatively small proportion of dead instructions, the aliasing is likely to be destructive), which can be estimated by the amount of code exercised by the benchmark relative to the predictor size. The predictors in Figure 5 use enough tag bits to guarantee no aliasing. We found that using 18 total bits (set index plus tag) gave nearly identical results on most benchmarks; accuracy was within 2% for all benchmarks with 16 total bits. Applications with larger code footprints would naturally require more bits to avoid aliasing.

3.2 Exploiting control flow information

The accuracy of the simple predictor can be improved by using control flow information just as the accuracy of branch prediction is enhanced by using information about prior branches (i.e., history). Unlike branch prediction, however, we want information about the *future* control flow because it uniquely determines the future path, and thus whether a particular value will be used.[†] At first glance, it may appear that this observation is not helpful since it requires ostensibly unattainable information about the future. However, we note that because of pipelining, control flow predictions generated in the earliest pipeline stages apply to the instruction stream *ahead* of the instructions being predicted by the dead instruction predictor. By bringing this information forward in the pipeline, we can enhance the accuracy of the predictions.

We studied two different methods of exploiting forward control flow information (referred to as a control flow *signature*). The simplest method uses the most recent N bits of global branch history as the signature. For N more than a few bits, some of the bits represent branches occurring before the instruction being predicted (the target instruction) and therefore this method also takes advantage of historical path information. The second approach uses only the predicted directions of those conditional branches between the front end of the machine and the target instruction. In this case, the signature also encodes the number of branch directions included (the *length* of the signature). If the first control instruction encountered after the target instruction is an indirect branch (including a procedure return), the branch’s target address is hashed into the signature instead. The two types of signatures are diagrammed in Figure 6.

Each predictor entry is augmented with a signature field. The signature becomes part of the tag for the predictor entry. Thus, there may be multiple entries in a set for a single static instruction differing only in signature. A predictor hit occurs when the stored control flow signature and address tag match the current signature and address tag exactly, corresponding to a match of both the instruction itself and its control flow context.

Figure 7 shows the performance of several predictors employing control flow information in the same format as Figure 5.

[†] The only case in which future control flow does not completely specify usefulness is when the value is contained in the source or destination register of a conditional move (`cmov`) instruction. We handle `cmov` instructions pessimistically, treating them as always using both sources and the destination register.

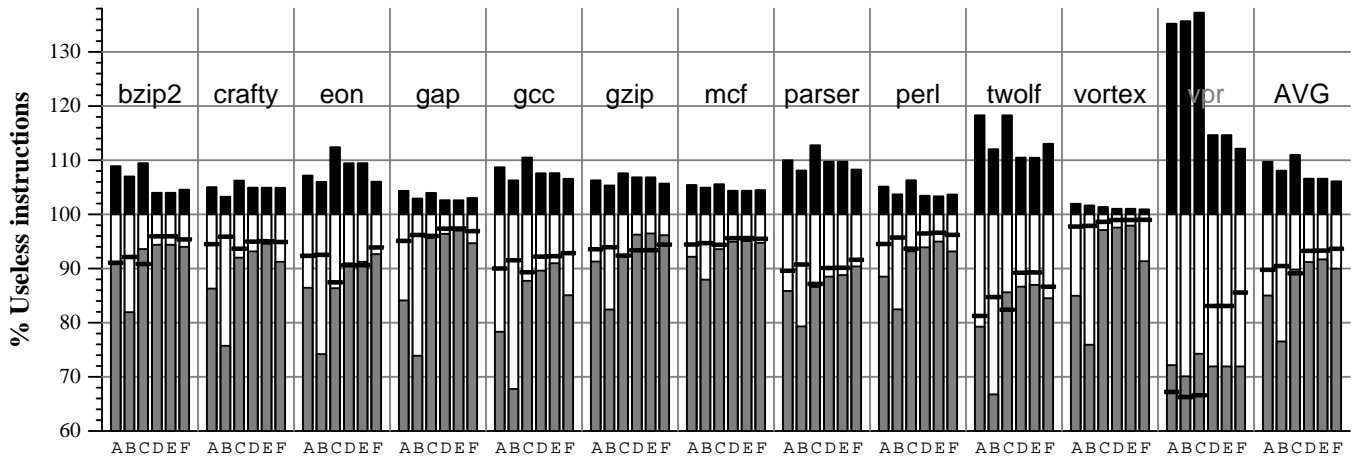


Figure 7. Performance of enhanced dead instruction predictors. This graph is identical to Figure 5 except for the predictor configurations. These are: (A) 5-bit branch signature; (B) 8-bit branch signature; (C) 5-bit future signature; (D, E) 5-bit future signature plus 2-bit confidence counter; (F) 5-bit branch signature plus 2-bit confidence counter. Configurations (A)-(D) are 2k-entry 4-way set associative with 9-bit tags; (E) and (F) are 2k-entry 8-way set associative with 10-bit tags.

Comparison with Figure 5 shows improvements in both accuracy and coverage over the simple predictor with the same organization. While the control flow predictors use more state, adding the same extra storage to the simple predictor hardly affects its performance. The data in Figure 7 plus data on other configurations (not shown) allow us to make the following observations. First, the accuracy of a predictor using a branch signature improves as more bits of branch history are used (A versus B); unfortunately, this is accompanied by significant decreases in coverage as the longer signatures are less likely to match exactly. A future signature of the same total length gives better coverage with slightly lower accuracy versus the branch signature (compare A and C). Longer future signatures (not shown) do not perform much differently since the pipeline depth limits the number of future branch predictions available to include in the signature—over 98% of instructions had fewer than four younger branches in the pipeline when they were renamed. Longer signatures only improve the predictor’s ability to distinguish indirect branch targets.

The predictors described thus far invalidate a predictor entry when that entry generates a misprediction. An alternative is to use a confidence mechanism [6]. One such mechanism adds to each predictor entry two bits that behave as a saturating counter. When that entry yields a correct prediction, the counter is incremented; on a misprediction, the counter is decremented. A prediction is generated only if the most significant confidence bit is set.

We found that the confidence mechanism increased both accuracy and coverage several percent (C versus D in Figure 7). The increase in accuracy was expected; the increase in coverage can be explained by recalling Figure 3(a): most useless instances come from instructions that usually produce useless instances. Therefore, the confidence mechanism allows a “mostly dead” instruction to continue to receive dead instruction predictions after generating the occasional useful result.

We also evaluated predictors employing alternative confidence policies (these results are not shown). Additional confidence bits resulted in slight accuracy gains at the expense of coverage. Requiring the confidence bits to be completely saturated in order to predict an instruction dead improved accuracy slightly but significantly decreased coverage. Replacing low confidence entries instead of the LRU did not noticeably change the results.

Finally, because control flow augmented predictors use multiple entries in one set for a single PC, the likelihood of conflicts is

increased. Therefore, we tested two configurations (E and F) that were the same total size, but were twice as associative. Comparing D and E shows that the increased associativity makes little difference for predictors employing a future signature. Comparing A and F shows a more significant effect for branch signature based predictors (most of the coverage difference is due to the increased associativity; all of the accuracy difference is due to the confidence mechanism).

3.3 Training and detecting mispredictions

In order to train the dead instruction predictor, the instruction stream must be observed during program execution. The use tracking table (UTT) is a small structure indexed by architectural register at the retire stage. Each UTT entry contains a flag indicating the usefulness of the value in the corresponding register, the PC of the instruction writing that register, and a path signature. As an instruction is retired, it sets the flags corresponding to its source registers. A value-producing instruction also creates a new entry, resets the flag corresponding to its destination register, and sets the PC and signature fields appropriately. The prior contents of the written entry are forwarded to a predictor write port.

A conceptually similar structure at the rename stage (the verification table or VT) may be used for early detection of mispredictions. In this case, the flag bit indicates whether the value in the corresponding architectural register is predicted to be dead. PC and signature fields are not required. If a use of a predicted-dead value is observed, a mispredict is signalled and recovery initiated. Due to wrong path execution, there exists a trade-off between how quickly a genuine misprediction is detected and the number of false mispredictions signalled. The earlier in the pipeline that the instruction stream is observed, the more likely that a wrong path use will result in a false misprediction. The best misprediction detection policy in any particular application depends on the cost of recovering from a mis-speculation and the accuracies of the dead instruction and branch predictors.

3.4 Hardware cost of prediction

Prediction overhead consists of the storage requirements of the predictor and the training mechanism. Our implementation (512-sets, 4-way set associative) employs a five bit control flow signa-

ture, two confidence bits, and a nine bit tag per entry, and five LRU bits per set resulting in a storage requirement of approximately 4.6 KB for the predictor. The UTT contains one flag bit, an eighteen bit PC field, and a five bit control flow signature for each architectural register. For our machine (thirty-one each integer and floating point registers), the UTT requires 186 bytes.

Neither structure is excessively multi-ported. Since a single access can retrieve entries for several consecutive PCs (the same order that the predictions are required), the predictor only requires one read port. A single separate write port suffices for predictor training, a low bandwidth operation. A single UTT read port is used to train the predictor table. The UTT requires a maximum of one read/write port per retiring instruction. In practice, this may be reduced since not all instructions produce a value.

Finally, we note that the access time constraint on the predictor structure is relaxed. Only the PC is required to access the proper table entry and this is available early in the pipeline. Thus, the predictor access can be pipelined to operate in parallel with the instruction cache access.

4 Useless Instruction Elimination

We now present a scheme for the elimination of the useless instructions identified by a predictor like that of the previous section. A high-level view of the integration of useless instruction elimination with a typical out-of-order pipeline is diagrammed in Figure 8. Elimination of predicted useless instructions occurs after renaming their source operands. The renamed instructions are kept in a new structure (the predicted useless table or PUT) until the speculation is verified. This significantly simplifies mis-speculation recovery; the only cost is the lost opportunity for reducing utilization of the rename structures. No destination physical register needs to be assigned. Prior to describing the elimination mechanism in detail, we discuss issues relating to validating predictions and handling loads.

4.1 Prediction verification

In this section, we examine properties of useless instructions that affect our ability to effectively eliminate them. In particular, we wish to address two questions: (1) how long does it take to detect and recover from a misprediction, and (2) how much speculative state must be maintained to validate a prediction.

The number of cycles between a value’s production and its first use can be used to estimate the time required to detect a mis-speculation of useless status. As soon as the first use of a predicted-useless value is observed, the recovery process can proceed. The

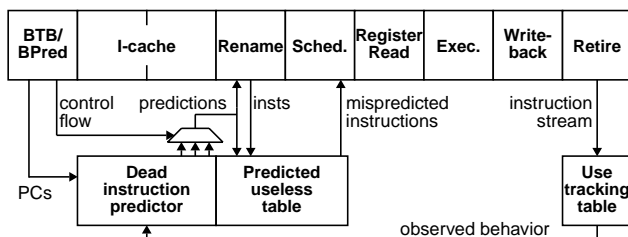


Figure 8. Processor pipeline with useless instruction elimination. The predictor is indexed by PCs from the front end; predicted control flow information is used to qualify the prediction. The predicted useless table stores eliminated instructions and monitors the rename instruction stream to detect mis-speculations. The use tracking table monitors the retirement stream to train the dead instruction predictor.

median distance between the rename of a value’s definition and the rename of its first use varies between 2 and 5 cycles. This statistic includes only those instructions that could potentially generate a useless instance. Thus, we can detect most incorrect predictions very quickly. We also note that only instructions data dependent on the mispredicted useless instruction are delayed by the recovery process. Furthermore, the delay is independent of the detection latency since the result of the mispredicted useless instruction is not needed until the dependent instructions are in the instruction window. Therefore, the costs of a mis-speculation will be low.

Only when all instructions between the prediction and the overwrite of the same destination have completed without a use of the predicted-dead value do we know that the speculation is correct. Thus, this distance (in instructions) serves as a measure of the amount of speculative state that must be buffered to ensure precise execution. The median number of instructions between consecutive definitions of an architectural register varies between 8 and 51 instructions. While the median def-def distance is low, we note that the tail of the distribution extends out to very long distances. Thus, in most cases, the verification of a prediction does not require a large quantity of speculative state, but there are a small number of useless instructions that we will be unable to verify within the limits of any reasonable reorder buffer size.

The amount of speculative state and complexity required to verify transitively useless status is increased substantially: *all* values derived from any instruction in the dependence tree must be over-written (killed) prior to their use by any instructions that are not themselves in the tree. Due to the limited potential benefit of handling these instructions and the increased complexity of their detection and verification, we do not attempt to detect or remove them.

4.2 Loads

Note that when a load is mispredicted to be useless, it must execute later as part of the recovery operation. If any intervening loads have been performed, this results in a reordering of loads that has implications for the memory consistency model. We do not address that issue except to note that the same solutions used to solve the analogous problem for any dynamically scheduled processor (in which loads can execute in arbitrary order) are also applicable in this case [12].

A more subtle problem with the elimination of loads arises when loads with certain side-effects are considered. Device drivers, for example, may depend on loads to certain memory-mapped hardware addresses. In this case, the side effect of the load (rather than the value loaded) may be the goal. Loads causing page faults or loads to intentionally out-of-bounds addresses are other examples of loads with side effects. We note that any load that could potentially be satisfied from the cache can not have a side effect. Therefore, a safe solution is to access the TLB for each eliminated load to ensure that the load address belongs to a cacheable page. A TLB miss results in a page fault and the scheduling of the load for execution. Using special instructions for those loads that have side effects also solves the problem, but requires a change to the ISA.

4.3 Useless instruction elimination

Figure 9 illustrates our scheme for the tracking and removal of useless instructions. Predictions are made in the rename stage of the pipeline by a dead instruction predictor (Section 3). The predicted useless table (PUT) is a small structure added to the rename stage to handle the elimination of useless instructions. Each entry

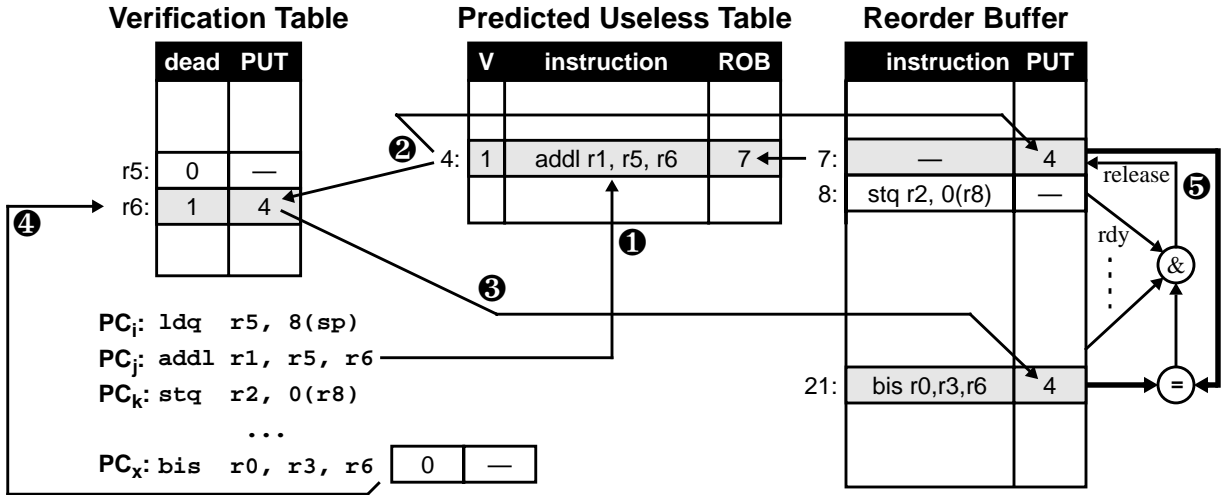


Figure 9. Removal of useless instructions. The value generated by the `addl` instruction at PC_j is predicted to be dead. ❶ After renaming the instruction’s source registers, it is placed into a free PUT entry. ❷ A pointer to this entry is placed into the VT and into a dummy ROB entry. ❸ When the verifying instruction at PC_x is renamed, it is annotated with the PUT field from the VT entry prior to ❹ installing its own data into the VT. The dummy instruction does not retire until ❺ the verifying instruction and all intervening instructions are ready to retire. At that point, the speculation is verified and the PUT entry is freed.

of the PUT contains a valid bit, a complete decoded instruction, and a pointer to a reorder buffer entry. Any instruction that is predicted to be dead is placed into a free PUT entry and the entry state is set to valid. The instruction’s sources are renamed normally, but no physical register is allocated for the result. The instruction does not get placed into the instruction window; instead, a dummy instruction with a pointer to the real instruction’s PUT entry is placed in the reorder buffer as a placeholder. A pointer to the PUT entry is also added to a new field in the VT entry corresponding to the destination register.

Misprediction recovery is straightforward. If the VT observes a use of a predicted useless value, the generating instruction is fetched from the PUT (via the VT pointer field), allocated a physical register, and inserted into the instruction window. The dummy reorder buffer entry is changed to reflect the instruction’s new status. If the underlying implementation demands the retirement of all in-flight instructions upon a context switch or interrupt, the PUT must be emptied into the instruction window in these cases as well. Otherwise, the PUT may be flushed along with all other in-flight instructions without impacting architected state.

The destination architectural registers of all instructions entering the rename stage are checked by the VT as part of its normal operation. When an overwrite of a predicted-dead value occurs and that value has a valid PUT pointer, the instruction being renamed is the *verifying instruction*. The PUT pointer is copied into a field in the verifying instruction’s reorder buffer entry. Before a placeholder instruction can retire, it must match its PUT pointer to that of a younger instruction in the reorder buffer (the verifying instruction). This matching is gated by the ready-to-retire status of each intervening instruction. When the verifying instruction and all older instructions are ready to retire, the placeholder retires and the corresponding PUT entry is freed. At that point, the instruction has been successfully eliminated. Note that the VT must be restored upon mis-speculations in order to tag the proper verifying instruction in the presence of wrong path execution. Fortunately the VT is small (6-7 bits per architectural register); keeping the state that must be restored to a minimum is the primary reason why the PUT and VT are separate structures.

Due to the finite size of the reorder buffer, we must be able to handle the case in which the verifying instruction does not appear in the instruction stream before the reorder buffer fills. In this case, recovery must be initiated as if there had been a misprediction to allow the unverifiable predicted useless instruction to retire. We call these missed opportunities *aborted predictions*. In practice, waiting until the reorder buffer is full before allowing the instruction to proceed carries an unreasonable performance penalty due to the resulting front end stall. A strategy that we have found to work well empirically initiates recovery (i.e., schedules the stalled instruction) if a predicted useless instruction is holding up retirement when the ROB reaches a threshold capacity. The selection of the threshold involves a trade-off between the number of instructions eliminated and the performance cost of waiting for verification. The selection of this threshold is addressed in Section 5.3.

5 Results

Prior to presenting our results, we describe our simulation methodology. All of the results contained in this section were generated by an execution-driven timing simulator of the Alpha architecture loosely based on SimpleScalar [1]. The benchmarks are the integer codes of the SPEC2000 suite compiled with the DEC compiler at the `-O3` optimization level as described in Section 2.1, except that we execute only the first 4 billion non-nop instructions. Our simulator eliminates nop instructions at fetch time; thus, they do not consume any resources, nor do they affect any statistics. We used configuration D from Figure 7 as the dead instruction predictor (512 sets, 4-way set associative with 9-bit tags and a 2-bit confidence counter per entry). Except for the sensitivity studies of Section 5.3, the ROB fill threshold for aborted predictions (see Section 4.3) is 224 (out of a ROB size of 256) and the PUT size is 32 entries. Table 2 summarizes other parameters of the simulated processor.

We evaluate our predictor against our goal of eliminating the costs of the useless instructions executed by optimized programs. Therefore, we should expect to detect and avoid the execution of as many of the available useless instructions as possible. As the

Table 2. Simulated processor parameters

Front-end	Instruction Window
4-wide fetch (nops skipped) with perfect BTB 12kB YAGS conditional branch predictor 64-entry return address stack 32 kB cascading indirect branch predictor	64 entries with 256 entry reorder buffer 4-wide issue, oldest ready first 320 physical registers (256 in-flight + 64 architectural) 4-wide retirement, except 2 stores per cycle maximum
Execution	Memory hierarchy
1 integer ALU (no multiply), 1-integer ALU (w/ multiplier), 1 branch unit 1-cycle latency all non-multiply insts, 3-cycle latency integer multiply 1 floating point ALU, 2-cycle latency 1 floating point ALU with multiplier/divider, 4/12 cycle latency mul/div 1 load unit, 3-cycle load to use latency on L1 hit, 1 store unit	64kB, 2-way L1I and D caches, 32-byte lines, perfect TLB's 2 MB, 4-way unified L2 cache, 64-byte lines, 6-cycle latency 64-entry unified prefetch/victim buffer 16-entry coalescing store buffer 80-cycle memory latency; opportunistic unit-stride prefetcher

execution of useless instructions unnecessarily increases resource utilization, we should expect measurable reductions in resource utilization. Finally, we should effect this reduction without impacting performance. Under resource constraints, useless instruction elimination should improve performance.

5.1 Resource Utilization

The data in Table 3 show that we are successful in eliminating 79% of the useless instructions accounting for 5% of all dynamic instructions. This is more than sufficient to significantly reduce resource utilization: register file and cache accesses, for example, are decreased by more than 5%. Four of the twelve benchmarks realize reductions of over 10% in one or both of register writes or cache accesses. The reduction in physical register allocations is comparable, reducing the effort required for free list management. Eliminated instructions do not consume execution resources, nor do they occupy instruction window slots (avoiding the expensive wakeup and select operations).

Differences in the relative reductions of executions, reads, writes, and allocations depend on the specific instruction mix of the benchmark. Benchmarks in which the useless instructions have proportionally more loads than the overall instruction mix (see Section 2.2) exhibit an amplified reduction in the number of cache accesses relative to the number of instructions eliminated (e.g., *twolf*). The mix of zero, one, and two input instructions affects the relative decrease in register reads and writes. Register writes are decreased relatively more than executions since not all instructions generate values (e.g., branches and stores), but all eliminated instructions do.

Figure 10 shows how each retired useless instruction was handled. On average, while 91% of actual useless instructions are identified by the predictor (this is basically the predictor coverage shown in Figure 7), only 79% of them are successfully eliminated. Three events prevent a predicted useless instruction from

being eliminated: (1) the PUT was full when the prediction was made, (2) the ROB fill threshold was exceeded before the prediction could be verified (aborted predictions), or (3) an apparent use of the value produced by the instruction was observed on a wrong path. Of these causes, the second is dominant, accounting for an average of 77% of non-eliminated correct predictions. The number of aborted predictions can be reduced by increasing the ROB threshold, although this can incur a performance cost (Section 5.3). Increasing the PUT size can address the first cause of aborted eliminations. Aborted eliminations due to wrong path uses depend on the structure of the program and the branch prediction accuracy.

5.2 Performance

Figure 11 shows how useless instruction elimination affects performance. The left bar for each benchmark corresponds to the baseline configuration (Table 2) while the right bar corresponds to the resource configuration of the Transmeta Crusoe [7], a more resource-constrained architecture. This architecture is identical to the baseline architecture except for the execution resources: one each of an integer ALU, a branch unit, an FP ALU, and a shared load-store unit. The figure also shows the performance of a perfect predictor and a perfect predictor required to verify its predictions (and therefore subject to the ROB threshold).

The data show that our scheme attains the reduction in resource utilization shown in Table 3 without a performance penalty. Eliminating useless instructions results in a 0.6% average speedup on our base architecture. The average speedup improves to 3.6% and ranges as high as 9.6% on the limited resource architecture.

A perfect predictor with verification gives performance almost identical to that of the real predictor, indicating that the cost of mispredictions is negligible. However, we note that prediction verification itself has a performance cost (represented by the size of the gray bar). While the ROB threshold prevents fetch stalls

Table 3. Utilization Impact of Useless Instruction Elimination

Benchmark	Useless insts eliminated	Reduction in executed insts	Reduction in register reads	Reduction in register writes	Reduction in register allocs	Reduction in D-cache reads
<i>bzip2</i>	88.15%	1.41%	1.41%	1.67%	0.96%	2.22%
<i>crafty</i>	84.82%	7.52%	8.75%	8.85%	7.62%	12.95%
<i>eon</i>	66.50%	3.21%	2.04%	4.65%	4.03%	3.93%
<i>gap</i>	89.82%	8.43%	8.21%	10.81%	10.39%	6.51%
<i>gcc</i>	79.49%	6.40%	6.92%	8.77%	7.26%	7.96%
<i>gzip</i>	88.11%	8.21%	10.98%	9.49%	8.18%	10.57%
<i>mcf</i>	82.30%	4.58%	5.23%	6.56%	1.96%	0.61%
<i>parser</i>	76.47%	3.74%	4.60%	4.86%	3.36%	4.64%
<i>perl</i>	82.98%	5.67%	5.73%	7.84%	5.89%	5.95%
<i>twolf</i>	72.25%	1.48%	2.03%	1.84%	1.42%	2.98%
<i>vortex</i>	77.81%	8.48%	7.78%	11.33%	11.15%	10.54%
<i>vpr</i>	61.26%	1.55%	1.31%	2.07%	1.05%	1.11%
Average	79.16%	5.06%	5.42%	6.56%	5.27%	5.83%

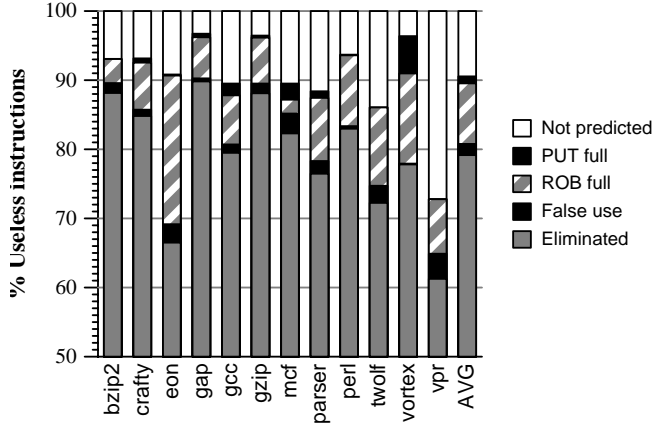


Figure 10. Retired useless instructions. This graph depicts the fraction of useless instructions that were eliminated and shows why the remainder were not. Causes of non-elimination are (1) no prediction, (2) no free PUT entry, (3) ROB threshold exceeded prior to verification, or (4) a wrong path use of the dead value.

for verification, the holdup of retirement delays training of the branch predictor (which happens at branch retirement time in our processor model). This results in a slight decrease in branch prediction accuracy, especially when many loop iterations containing a mispredicted branch occur immediately after an eliminated instruction.

Although we achieve a modest performance improvement under resource contention, we believe a greater performance advantage can be realized by freeing a compiler to optimize more aggressively, particularly when code motion is involved. While the implementation and evaluation of alternative compiler algorithms is beyond the scope of this work, we note that existing resource-aware optimization algorithms (e.g., instruction scheduling itself or the algorithms for partial dead code and redundancy elimination proposed by Gupta et al. [5]) can benefit from ignoring the resource costs of hoisted instructions on the paths where their results will be unused.

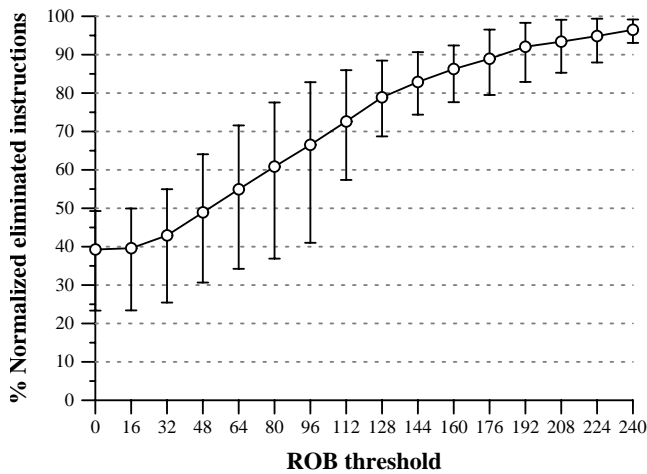


Figure 12. ROB fill threshold sensitivity. Useless instructions eliminated as a function of ROB threshold normalized to a threshold equal to the ROB size. The results for the highest and lowest benchmark are shown in addition to the average.

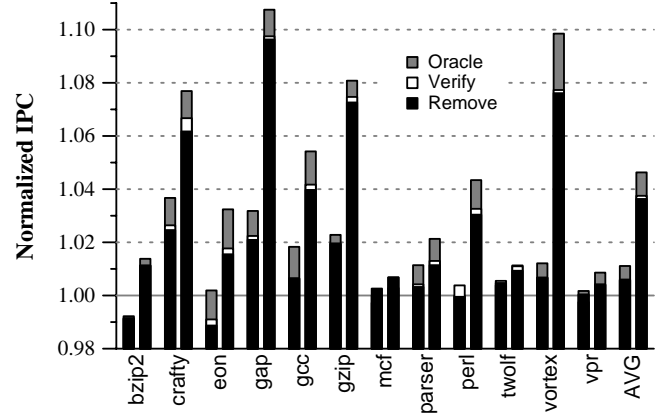


Figure 11. Performance of the baseline predictor (black), a perfect predictor required to verify predictions (white), and a perfect predictor with no verification cost (gray). Base case is the same architecture without useless instruction removal. Two configurations differing in execution resources are shown.

5.3 Sensitivity Analysis

Figure 12 shows the number of eliminated instructions as a function of ROB threshold. The results are normalized to a threshold equal to the ROB size (256). The results for the minimum and maximum benchmarks are shown as well as the average over all benchmarks. A zero threshold means that verification must be possible immediately when the predicted useless instruction reaches the head of the ROB. A higher ROB threshold results in more eliminated instructions, but as alluded to in Section 4.3, has an associated performance cost. The number of additional useless instructions captured increases more slowly than the maximum potential retirement backup. Thus, at some threshold value, the increasing cost of the retirement backup exceeds the benefit of the small number of additional instructions eliminated. Performance results (not shown) indicate that the peak average performance occurs at a threshold of 144 although the minimum performance benchmark continues to improve to a

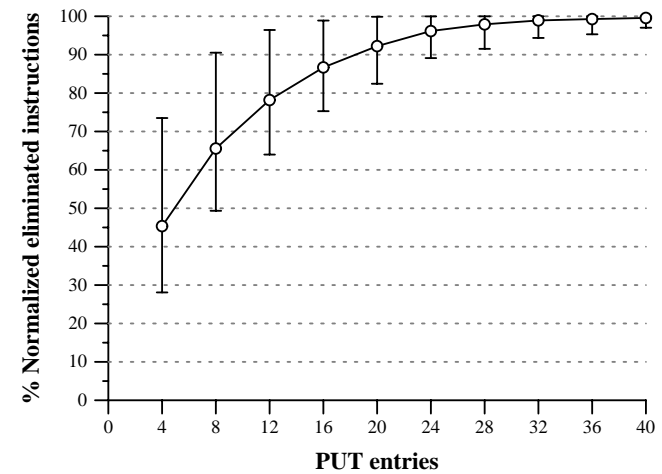


Figure 13. PUT size sensitivity. Useless instructions eliminated as a function of PUT size normalized to an infinite PUT. The results for the highest and lowest benchmark are shown in addition to the average.

threshold of 192. We chose a threshold of 224 to maximize the reduction in resource utilization. The average performance difference between thresholds of 224 and 144 is less than 0.15%.

Figure 13 is similar to Figure 12, but shows the effect of the PUT size on the number of eliminated instructions. The results are normalized to an infinite PUT. Little additional benefit is achieved for PUT sizes beyond 32 entries. The only cost of an increased PUT size is the hardware overhead.

6 Related Work

There has been an extensive amount of work performed on characterizing and eliminating dead instructions. Partial dead-code elimination [8] is a compiler algorithm that transforms code to reduce or eliminate instances of instructions that produce dead values. In essence, the algorithm detects code that generates values used on only some subsequent control flow paths and attempts to move that code down into those paths.

Martin et al. proposed a cooperative software/hardware scheme to track registers containing dead values [10]. Their scheme involves annotating the executable with information from the compiler about the last use of register values (i.e., noting that specific registers are dead). This information is subsequently used by the hardware to enable early physical register reclamation and elimination of needless saves and restores across procedure calls and context switches. Our scheme does not eliminate the saves of dead values (since we execute all stores). However, we do not require cooperation by the compiler or modification of the executable and we are also able to eliminate useless instructions within procedures.

Yoaz et al. [13] also observed the occurrence of dynamically dead instructions, focusing on the subclass of dead instructions called silent stores [9]. They mentioned the possibility of squashing or de-prioritizing dead instructions, but did not present any specific schemes to identify or handle them.

Rotenberg proposed exploiting sequences of instructions that have no externally visible effects [11]. These ineffectual instructions include dead instructions, silent instructions (stores and otherwise), correctly predicted branches, and instructions transitively connected only to other ineffectual instructions. He proposed skipping these instructions to allow a speculative thread to get ahead of a slower verification thread.

7 Conclusions

In this paper, we examined the costs associated with compiler optimizations. Partially dead code, introduced by optimizing compilers during inter-block instruction scheduling, results in the execution of a large number of useless instructions. Although these additional instructions may not penalize performance (except under resource constraints), they have other processing costs associated with them (e.g., utilization of resources and power consumption). These costs can prevent compilers from making optimizations that might have a performance benefit.

We showed that useless instructions exhibit locality that makes them amenable to prediction: most useless instructions arise from a small number of static instructions and these are likely to produce useless instructions most of the time. Then, we presented a highly accurate predictor for the usefulness of an instruction. Our predictor design achieves 93% accuracy and detects 91% of the useless instructions in our benchmarks with less than 5 KB of storage. An important novel contribution was the use of future control flow information in the form of predictions generated for younger instructions in the pipeline.

We then presented a scheme to avoid the execution of predicted-dead instructions, alleviating the costs associated with these instructions. Using this scheme, our processor model was able to avoid the execution of 79% of the useless instructions that were present across our benchmarks. The resulting reductions in utilization of the data cache and register file averaged between 5 and 7% and in several cases exceeded 10%. These significant reductions in the costs of useless instructions were achieved without a performance impact due to mis-speculation. On a resource-constrained architecture, we demonstrated performance improvements of up to 9.6%.

Acknowledgements

The authors would like to thank Brandon Schwartz, Brian Fields, Paramjit Oberoi, Craig Zilles, and Milo Martin for their helpful comments on early versions of this work. We are especially grateful to Craig Zilles for supplying his simulation infrastructure.

This work was supported in part by National Science Foundation grants EIA-0071924 and CCR-9900584 and the University of Wisconsin Graduate School. Adam Butts is supported by a fellowship from the Fannie and John Hertz Foundation.

References

- [1] D. Burger and T. Austin. The SimpleScalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin-Madison, June 1997.
- [2] P. Chang, N. Warter, S. Mahlke, W. Chen, and W. Hwu. Three architectural models for compiler-controlled speculative execution. *IEEE Trans. on Computers*, 44(3), March 1995. pp. 481-94.
- [3] W. Chen, S. Mahlke, N. Warter, S. Anik, and W. Hwu. Profile-assisted instruction scheduling. *Intl. Journal for Parallel Programming*, 22(2), April 1994. pp. 151-81.
- [4] K. Cooper, M. Hall, and K. Kennedy. Procedure cloning. In *Proceedings of the IEEE 1992 Intl. Conference on Computer Languages*, April 1992. pp. 96-105.
- [5] R. Gupta, D. Berson, and J. Fang. Resource-sensitive profile-directed data flow analysis for code optimization. In *Proceedings of the 30th Annual Intl. Symp. on Microarchitecture*, December 1997. pp. 358-68.
- [6] E. Jacobsen, E. Rotenberg, and J. E. Smith. Assigning confidence to conditional branch predictions. In *Proceedings of the 29th Intl. Symp. on Microarchitecture*, December 1996. pp. 142-52.
- [7] A. Klaiber. *The technology behind Crusoe™ processors*. Transmeta Corporation White Paper, January 2000.
- [8] J. Knoop, O. Ruthing, and B. Steffen. Partial dead code elimination. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, June 1994. pp. 147-58.
- [9] K. Lepak and M. Lipasti. On the value locality of store instructions. In *Proceedings of the 27th Annual Intl. Symp. on Computer Architecture*, June 2000. pp. 182-91.
- [10] M. Martin, A. Roth, and C. Fischer. Exploiting dead value information. In *Proceedings of the 30th Annual Intl. Symp. on Microarchitecture*, December 1997. pp. 125-35.
- [11] E. Rotenberg. Exploiting large ineffectual instruction sequences. Technical Report, North Carolina State University, November 1999.
- [12] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2), April 1996. pp. 28-41.
- [13] A. Yoaz, R. Ronen, R. Chappell, and Y. Almog. Silence is golden? Presented at the *7th Annual Symp. on High Performance Computer Architecture*, January 2001.