Characterizing and Predicting Value Degree of Use

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

Abstract

A value's degree of use*—the number of dynamic uses of that value—provides the most essential information needed to optimize its communication. We present simulation results demonstrating the properties of degree of use of values, including their predictability: most static instructions generate values with few degrees of use and these exhibit temporal locality. We use these results to guide the design of a degree of use predictor. The development and detailed characterization of this predictor is the focus of this paper. Our predictor leverages future control flow information (e.g., branch predictions) to select among different possible degrees of use. We study the effects of several optimizations and variations in the predictor's algorithms to tune the predictor for maximum performance. The resulting predictor generates correct degree of use predictions for over 92% of all dynamic values and has a misprediction rate below 2.5%. Such a predictor has a wide range of potential applications in optimizing value communication.*

1 Introduction

Inter-instruction communication is a critical challenge today, and it is likely to be a significant challenge facing future processor and system designs. In exploiting instruction-level parallelism, current processors expend considerable resources (*e.g.*, complicated register files, bypass networks, and instruction wakeup logic) to enable adequate inter-instruction communication bandwidth at low latencies. These resources typically dwarf those used to actually execute instructions and represent the primary constraint on frequency scaling in future designs [3, 7]. Increasing demand for ever more capable communication networks calls for designs optimized for the required value communication. Knowledge of value communication is the key to designing and making efficient use of novel communication structures.

There are two aspects of value communication: (1) the number of, and (2) the identity of each value's consumers. In this paper, we focus on the first aspect—the number of consumers of a value, and we restrict our attention to values communicated through registers. We define *degree of use* to be the number of times a particular value is used by all subsequent instructions in the program. A value's degree of use is a direct indicator of its communication resource requirements, an essential part of any scheme for optimizing communication.

We propose gaining knowledge about a value's degree of use with a *degree of use predictor*. The demonstration of accurate degree of use prediction is the primary focus of this paper. We find that most static instructions produce values with a unique degree of use; those that produce values with varying degrees of use exhibit a small degree of use working set, making even these readily predictable. We propose a degree of use predictor and evaluate a number of enhancements to the basic design. Our optimized predictor achieves over 97% accuracy averaged across our benchmarks while generating correct predictions for 92% of dynamic values. Storage requirements are modest (the preceding numbers refer to a predictor using only 8.5 KB).

Knowing the number of consumers of a dynamic value before that value is even generated enables several potential optimizations to the value communication structures (*e.g.*, the instruction window, bypass networks, and register file). We suggest some of these applications in the next section, one of which has been evaluated previously [2], but we do not evaluate a specific application in this paper.

The remainder of the paper is organized as follows. We present the methodology we used to gather the results in the remainder of the paper in Section 3. Data on the behavior and predictability of value degree of use appear in Section 4. We present novel information showing that degree of use is a predictable property of dynamic executions. Section 5 develops and evaluates a degree of use predictor in detail, including discussions on training, prediction verification and misprediction detection. We discuss future work and conclude in Section 6.

2 Applying degree of use in communication optimization

We have identified numerous possible applications of accurate degree of use prediction, some of which we describe in this section. While the focus of this paper is on the prediction of degree of use, we would be remiss if we did not motivate the predictor with potential applications. Proving that accurate degree of use prediction is feasible is a prerequisite for all of the optimizations described below. Thus, we do not evaluate any specific application in this paper. However, it is important to real-

ize the variety and potential impact of the applications enabled once a predictor is available.

We recently proposed a method for the detection and elimination of dynamically dead instructions (*i.e.*, those generating degree of use zero values) to enable more aggressive compiler optimizations [2]. The dead instruction predictor proposed in that work is a special case of a degree of use predictor, allowing the same optimization to be realized with a degree of use predictor instead. Using some of the techniques we describe in this paper, the performance of the generalized predictor can actually exceed that of the dedicated predictor. The degree of use predictor can also be leveraged to perform other optimizations simultaneously.

Others have noted [5] and we will show in Section 4 that most values are used only once before being overwritten. Given the abundance of degree of use one values, mechanisms exploiting this phenomenon should be widely applicable. Regarding the actual communication of these values, it is obvious that use of the register file results in unnecessary overhead. The register communication model implicitly (but incorrectly) suggests that a value bound to a register will be used multiple times. Values with a predicted degree of use of one need not even use a register. Instead, the communication of these values can occur entirely through the bypass network with proper attention to the scheduling of the producer and consumer operations. No registers need even be allocated to these values. The combination of the reductions in the number of registers required and the number of register file write ports could significantly reduce the size of the register file, allowing it to be faster and/or lower power.

The scheduling of dependent operations themselves could also be simplified with knowledge of degree of use. Instructions with a predicted degree of use of one can be allocated dedicated reservation stations that are directly addressable by the completing parent instruction. The dependent instruction can be steered to this reservation station by information available at the rename stage. Upon completion of the parent instruction, the wakeup operation would not require a tag broadcast across a large associative instruction window. Instead, the proper dependent operation could be woken up directly.

The knowledge that the value communicated between two instructions is private (*i.e.*, has a degree of use of one) can also be exploited to dynamically collapse dependent operations. Given simple enough operations (*e.g.*, dependent logical operations), it is quite possible to complete both operations in a single cycle. The resulting reduction in the dataflow height could result in increased performance. Interlock collapsing ALUs have been proposed as a means of executing two dependent operations together [8], but the application of this technique is limited by the need to statically ensure that the dependent operation is the only consumer. A degree of use predictor can increase the applicability of this technique by identifying such instances dynamically.

Other optimizations can exploit degrees of use greater than one. Consider an instruction processing back end that is optimized for the execution of strands of dependent operations that communicate no intermediate values (*e.g.*, the instruction level distributed processor [6]). Instructions that generate values with low degrees of use may be duplicated such that each instruction generates a value used only once, reducing the number of values that have to be communicated globally. Global communication is costly in distributed and clustered architectures and redundant execution may actually be cheaper than communication for these designs.

We will show that accurate degree of use predictions can be delivered at a very small cost. The exploitation of degree of use prediction to optimize value communication is part of our ongoing work. We reiterate that in this work we focus on the characteristics of degree of use and the predictor itself, abstracting the development of the predictor from the details of a particular application. The development of the predictor is an essential first step in developing any of the applications more fully.

3 Methodology

The results in this work were generated using execution-driven simulators of the Alpha ISA loosely based on SimpleScalar [1]. The degree of use characterization data in Section 4 was generated using a purely functional simulation. Due to the effect of the pipeline structure on degree of use prediction a detailed timing simulator was needed to properly evaluate the degree of use predictors in Section 5. Both simulators eliminate nop instructions at fetch time; consequently, nops do not contribute towards any of the data presented herein. We do not track uses of values through memory or across system calls. A system call is treated as a use of every architectural register. In the predictor studies of Section 5, a system call completely flushes the predictor state to account for pollution occurring during a context switch. Uses of a register following a system call prior to any other definition of that register are not attributed to any instruction. Stores are handled as a single use of the register containing the stored value while each load is treated as though it creates a new value. Additional details on the timing simulator are listed in Table 1.

Our benchmarks are the integer codes of the SPEC2000 suite. All of the benchmarks were compiled with DEC C V5.9 or DEC C++ V6.1, as appropriate, and statically linked. Optimization options were `-arch ev6 -fast -O3`. We run each benchmark to completion on the train inputs with the exception of `vpr`, `eon`, and `perl`, all of which specify multiple runs with different inputs as part of the train input set. We ran only the routing phase of `vpr`, the kajiya input to `eon`, and the diffmail program with `perl`.

Table 1. 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**
3 integer ALUs, 1-cycle latency 1 integer multiplier, 3-cycle latency 2 floating point ALUs, 2-cycle latency 1 floating point multiplier/divider, 4/12 cycle latency 2 load-store units, 3-cycle load to use latency on L1 hit	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

4 Characterizing degree of use

Figure 1 illustrates some of the interesting properties of degree of use with a short code example and a portion of the resulting dynamic dataflow graph. First, we note that many instructions generate a single, unique degree of use. The values assigned to `t2` and `t3`, for example, will always exhibit a degree of use of one. The values in these registers do not correspond to actual source variables, but are temporaries inserted by the compiler. Such temporaries frequently exhibit a degree of use of one.

Values with live ranges that span conditional branches can have different degrees of use depending on the particular path taken through the program. The value in `t1` has a degree of use of one in the final loop iteration, but a degree of use of three in all prior iterations. If the outcome of the loop terminating `beq` instruction were available before the `ldq` executed, then degree of use of the value would be known when it was generated. We will use this observation in the design of our predictor in Section 5.2.

Finally, we note that it is possible for a value to have a degree of use of zero (*i.e.*, never be used). Had the loop terminated due to the `bge` being taken, the two shaded nodes in the dataflow graph would not be executed and the value loaded by the `ldq` instruction would not be used. The instructions generating such values have been examined in detail in our prior work [2].

4.1 Degree of use characteristics

Figure 2(a) shows data on the observed degree of use for the benchmark programs. The bars in the graph represent total number of value-producing instructions. These account for 75.4% of all (non-nop) retired instructions on average; the remainder are stores and branches that do not generate register values. It is readily apparent that most of the communication occurring during program execution is direct communication as 65% of dynamic values have degree of use one. Degree of use two values account for only 14% of all values. No higher degree of use accounts for more than 7% of the values in any of the benchmarks. We also observe non-negligible numbers (9.2% on average) of dynamic values with a degree of use equal to zero.

While the number of values with a high degree of use is very small, the degrees of use themselves can be large. Maximum degrees of use range from about ten thousand to over three hundred million (initialization of the global

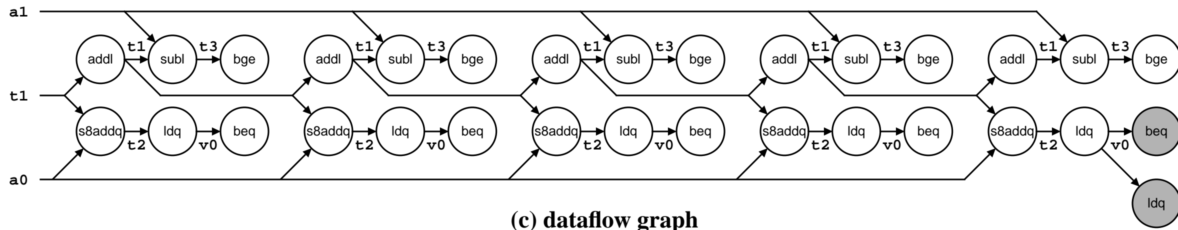
```
do {
      bucket_ptr = hash_table[idx];
      idx++;
      if (idx >= num_buckets) return 0;
} while (bucket_ptr == NULL);

return bucket_ptr->list_head;
```

(a) source code

```
loop: s8addq t1, a0, t2
      ldq    v0, (t2)
      addl   t1, 0x1, t1
      subl   t1, a1, t3
      bge    t3, exit
      beq    v0, loop
      ldq    v0, 24(v0)
      ret    zero, (ra), 1
exit: bis    zero, zero, v0
```

(b) assembly code



(c) dataflow graph

Figure 1. A sample code sequence in (a) source code and (b) assembly and (c) the corresponding dataflow graph. This loop finds the first occupied bucket in a hash table. The dataflow graph corresponds to five loop iterations and the loop exit. Each node represents a dynamic instruction while the arcs represent values communicated among the instructions. The exact number of consumers of a value is not known until the architectural register is reassigned to a different value since subsequent instructions then have no way to name the old value. Note that many of the dynamic values are used by only one consumer instruction.

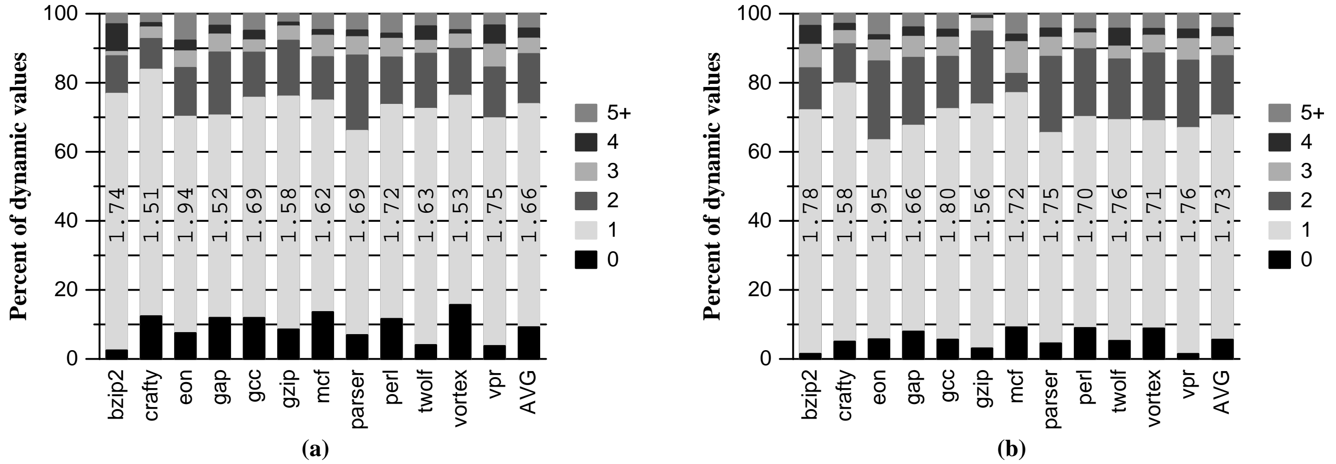


Figure 2. Observed degree of use. Number of uses of each dynamic value generated during the execution of each benchmark. The average degree of use appears superimposed on each bar. Benchmarks compiled with (a) DEC C/C++; (b) gcc/g++.

pointer in `bzip2`) across the different benchmarks. Instructions generating the largest degrees of use fell into two overlapping categories: (1) base address generating instructions (often stack and global pointer updates), and (2) instructions generating loop-invariants. In the first case, the number of unique static consumers tends to be high, while in the second case the high degree of use frequently results from repeated communication to a small set of static consumers. Both of these classes of instructions are illustrated in the example of Figure 1. The values in `a0` and `a1` are both loop-invariants; the value in `a0` is also a base address (of the array `hash_table`).

The degree of use data seem largely compiler independent. Figure 2(b) shows the same data for the benchmarks compiled with gcc/g++.† The only trend that is immediately obvious is that the gcc compiled benchmarks generate significantly fewer degree of use zero instructions (65% fewer on average). The frequency of degree of use zero instructions is highly sensitive to how aggressively the compiler optimizes the code [2]. Of these two compilers, the DEC compiler performed more extensive optimizations (*e.g.*, hoisting instructions above conditional branches), resulting in increased number of degree of use zero values. The average degree of use differed only 4% between the two compilers. This should not be surprising since the overall value communication structure is specified by the ISA and the program itself. Minute differences arise from differences in how the compilers perform register allocation, code scheduling, and other optimizations.

Franklin and Sohi previously investigated degree of use and register lifetimes [5]. Their goal was the optimization of the register file. In spite of using a different instruction set and compiler and different benchmarks, their data on degree of use are quite similar to that we present here. While they presented aggregate degree of use data, they did not delve into the per-instruction behavior or the predictability and application of degree of use. Recently, Eeckhout and Bosschere observed that the relative frequencies of values with different degrees of use were well-described by a power law model [4].

† Version 3.0.3 of the gcc compilers were used except that version 2.95.3 was used to compile the `gcc` benchmark because of a bug. Compiler options were `-mcpu=ev6 -O3`.

4.2 Predictability of degree of use

The predictability of degree of use has two components: (1) the working set of value-producing instructions, and (2) the working set of the degrees of use generated by any particular instruction. For a finite-sized predictor, the first component determines the *availability* of a prediction for a particular instruction (referred to as coverage). For a fixed coverage, a larger working set of value producing instructions demands a predictor with more capacity. In this sense, the coverage of a degree of use predictor is analogous to the hit rate of a standard cache. The working set of degrees of use determines the difficulty of providing a *correct* prediction: when instances of a particular static instruction generate values with many different degrees of use (*i.e.*, a large degree of use working set), selecting the correct degree of use becomes a problem.

We note that the working set of value-producing instructions is related to instruction cache performance. Low I-cache miss rates indicate working sets of instructions that fit within the cache. However, unlike the instruction cache, a degree of use predictor need not have state for every executed instruction. Branches and stores produce no register values; eliminating these instructions reduces the number of static instructions that must be stored in a predictor. Additionally, a degree of use predictor need not be constrained by the instruction cache architecture (*i.e.*, its size and associativity). Figure 3 shows the coverage that would be achieved if predictions were available for the last N unique instructions. As one might expect, the number of entries required for a given coverage is correlated with the text size. The data show that a 4k-entry fully-associative predictor achieves at least

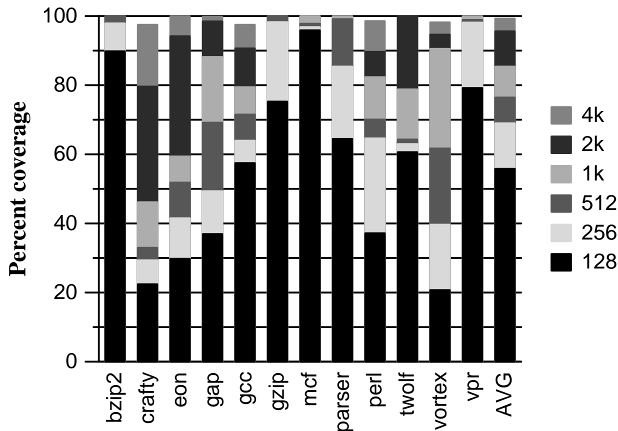


Figure 3. Number of instructions resulting in a given predictor coverage. This distribution shows the percentage of instructions for which a prediction would be available (coverage) given a fully-associative predictor of the given size with LRU replacement.

97% coverage on all benchmarks. A direct mapped predictor of the same size will likely have lower coverage due to the potential for conflict misses in the predictor.

Even more important to the feasibility of accurate degree of use prediction is the stability of the degree of use among values generated by the same static instruction. Figure 4 depicts the distribution of static instructions by the number of unique degrees of use generated. The majority of static instructions (84% on average) generate the same degree of use every execution. We also observe that the degree of use is extremely stable, even for those instructions that produce values with multiple degrees of use. Figure 5 shows a distribution of all dynamic values by how recently a value of the same degree of use came from the same static instruction. In other words, if the last N unique degrees of use for each static instruction could be remembered, Figure 5 shows how often the degree of use for the next value would occur in this set. The data show that simply remembering the last degree of use generated by every static instruction is sufficient to give 93% accuracy. If a means were available to distinguish between the last two degrees of use, that figure increases to almost 99%.

5 Degree of use prediction

In order to exploit degree of use in communication optimization, the information must be available by the time the value is produced. Once the actual uses of a value are observed, few, if any, optimizations may be made. Prediction solves the problem of getting the needed information in a timely fashion provided the predictor is located early enough in the pipeline and has adequate accuracy.

Degree of use prediction is fundamentally different from most other types of prediction in use today. The root of this difference is that the instruction that verifies a prediction is *not* that to which the prediction applies. Contrast this with branch or value prediction: for these techniques, the same instruction that initiates a prediction validates it. However, the final degree of use of a value is only certain when the value is overwritten by a completely unrelated value. Any instruction that reads the value between its generation and its destruction affects the actual degree of use and therefore must be tracked. The consequences of this difference affect predictor training, prediction validation, misprediction detection, and the structure of the predictor itself.

All of these aspects of degree of use prediction will be addressed in this section. We begin with a description of how degree of use is determined since this is necessary

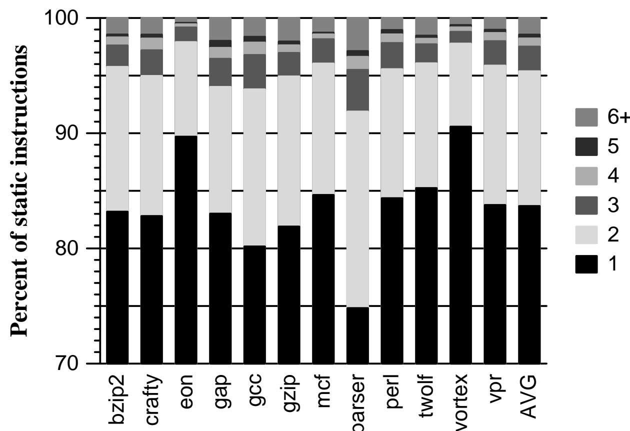


Figure 4. Unique degrees of use. This distribution shows the number of unique degrees of use generated by each static instruction over the entire execution of the benchmark. Note the offset of the y-axis.

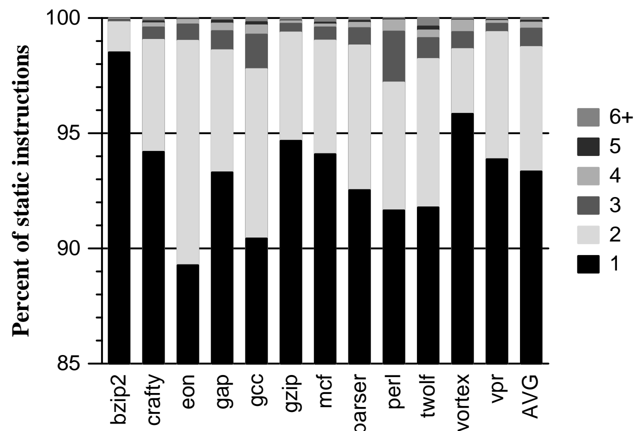


Figure 5. Stability of degree of use. This distribution shows the working set size of degrees of use per static instruction. The "1" bars count dynamic degrees of use equal to that generated immediately previously by the same static instruction. Similarly, the "2" bar means the degree from an instance of the static instruction was not the most recently seen (regardless of how many times in a row that degree had been generated) but the prior unique degree. Note the offset of the y-axis.

for both training and evaluating prediction outcomes. We then use the properties of degree of use outlined in the previous section to guide the development of an accurate predictor that we then evaluate in detail. We present several different predictor enhancements that improve the accuracy, coverage, and storage efficiency of the basic predictor. Then, we revisit the issue of observing the degree of use in the context of prediction verification and misprediction detection.

5.1 Observing degree of use dynamically

The determination of degree of use is necessary both to train a predictor and to verify the accuracy of the predictions that are generated. Fortunately, calculating the degree of use is straightforward using the in-order instruction stream at retirement. For each architectural register, one maintains a counter that is incremented when a use of the corresponding register is observed. When a register is overwritten, its counter indicates the degree of use of the value previously in the register. The counter is reset and the process resumes.

We refer to this set of counters as the *degree training table* (DTT). The counters saturate at the maximum predictable degree of use, a limit discussed in Section 5.5. In addition to the counter, each entry contains information about the dynamic instruction that produced the value currently in the corresponding register (*e.g.*, that instruction's PC). This information is used with the final degree of use in training the predictor. Each entry also contains the predicted degree of use for that value (if any).

The operation of the DTT is illustrated in Figure 6. As an instruction becomes ready to retire, it increments the use counters corresponding to its source registers. A value-producing instruction also overwrites the entry corresponding to its destination register. The entry is modified by resetting the use counter and setting the instruction identifying fields and the predicted degree of use appropriately. The prior contents of the written entry are forwarded to the predictor for training.

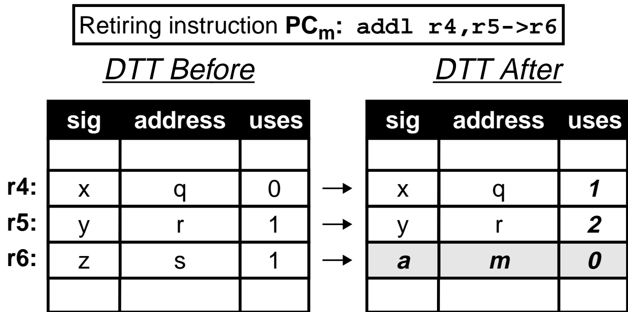


Figure 6. Structure and use of the degree training table. The retirement of the instruction at PC_m increments the use counts of the source registers and updates the entry for the destination register (shaded). The **sig** fields contain the control flow signature used during the degree prediction of the corresponding instruction (see Section 5.2). The prior contents of the entry for **r6** are sent to the degree predictor for training.

The DTT is also used for prediction verification. Note that a value must be overwritten before its final degree of use is certain (since additional uses may occur as long as the value is available). At the time of the overwrite, the observed degree of use is compared to the predicted degree of use to verify the prediction. Underpredictions (when more uses occur than were predicted) can be signalled immediately when the predicted use count is exceeded (prior to the overwrite of the value).

5.2 Developing the predictor microarchitecture

The data in Figure 4 suggest that the identity of the generating instruction is critical to determining a value's degree of use since the majority of instructions generate a unique degree of use. Furthermore, Figure 5 demonstrates that static instructions that generate multiple degrees of use demonstrate temporal locality in the degrees of use that they cause. The combination of these characteristics indicates that a cache-like predictor indexed by the PC of each value producing instruction will be successful. Indexing the predictor table by the instruction PC also allows the predictor to be queried early in the pipeline where the predictions can be used to maximum advantage. A basic degree of use predictor could simply associate the last observed degree of use for each static instruction.

While the data in Figure 5 suggest that the performance of such a basic predictor would be reasonable, we have the opportunity to do much better given a means of distinguishing among multiple degrees of use for a single static instruction. This requires additional information to be used in the generation of a prediction. Inspired by the success of branch prediction schemes that employ various forms of branch and path history, we investigated the application of control flow information to degree of use prediction. The degree of use of a value is completely determined by the instructions that are encountered after the value is generated. Because these instructions are in turn determined by the future control flow, the future control flow is the ideal information to discriminate among different potential degrees of use for a value.

In our prior work on identifying dynamically dead instructions, we observed that due to pipelining, instructions in the middle of a pipeline have available to them the predicted outcomes of control instructions that occur *later* in the dynamic instruction stream [2]. Thus, if the selection of a final degree of use prediction can be delayed several pipeline stages, the predicted directions of many subsequent branches will be available for use in making the prediction. Given a high branch prediction accuracy, these predictions are equivalent to path lookahead, providing nearly perfect knowledge about the uses of a value that will be observed in the immediate future.

We employ the *control flow signature* as described in our prior work to exploit future control flow information. The signature stores the number and predicted directions of conditional branches between the instruction for which the prediction is being generated (the

target instruction) and the front end of the pipeline. If an indirect branch (*e.g.*, a procedure return) occurs before any conditional branch, the signature instead contains bits from the predicted target address. The encoding we use sets the most significant bit if the remaining bits encode an indirect target address and clears this bit if the signature contains conditional branch directions. In the latter case, the highest set bit indicates the number of branch directions contained in the lower order bits. Initially, we require that a stored signature exactly match the current signature (maintained by the front end) in order to supply a prediction. We will explore an alternative policy in Section 5.4.

In order to maximize the lookahead possible using a control flow signature, the signature should be generated as late as possible in the pipeline (to allow more instructions to contribute). Thus, there is a tension between waiting for more information to use in generating a prediction and obtaining the prediction as early as possible in the processing of the target instruction. Because the fidelity of a signature increases the later it is generated, the selection of a prediction based on the signature should be the final step in generating a prediction. Fortunately, the target instruction's PC is available much earlier to initiate the prediction, allowing the latency of accessing the predictor table to be overlapped with earlier pipeline stages.

Two factors are important in determining the final deadline for the prediction. Since the prediction will be used for optimizing value communication, it should be available before any resources used in value communication (*e.g.*, physical registers, reservation stations, register write ports) are allocated. Also, since the predictor accesses are initiated in order, the predictions should be consumed at an in-order stage of the pipeline to simplify the matching of instructions with their predictions. Both of these requirements are met by making the predictions available in the rename or allocate stage of the pipeline. One of these two stages is usually the last pipeline stage before the dynamically scheduled processor core, allowing for the longest possible control flow signature as well.

The resulting predictor organization is shown in Figure 7. Instruction PCs from the front end are used to access the predictor table in parallel with the fetching of the instructions from the I-cache. The current control flow signature (communicated forward from the front end) is used to select the final prediction in the rename stage (our pipeline does not have an allocate stage and performs resource allocation during rename). The predictor table is organized as a set-associative cache. Each entry contains a tag and control flow signature to use in selecting the final prediction and the predicted degree for that entry. The maximum encodable degree is reserved to indicate an invalid entry while the highest *valid* prediction is assumed to mean a degree greater than or equal to its actual value (*e.g.*, for a three-bit degree field a value of six represents a predicted degree of six or greater and seven denotes an invalid entry). For our evaluations, we use a three bit degree field; we discuss the choice of the maximum degree limit in Section 5.5.

5.3 Predictor evaluation

Evaluating a degree of use predictor is complicated by two factors. First, the prediction can take on many values, only one of which is correct. Thus, an incorrect degree of use prediction may be an underprediction or an overprediction of the actual degree of use. Second, predictions may not be generated for all values. To simplify our discussion, we employ two metrics of overall predictor performance. Coverage is defined as the percentage of all values for which a prediction (right or wrong) is generated, while accuracy equals the percentage of all

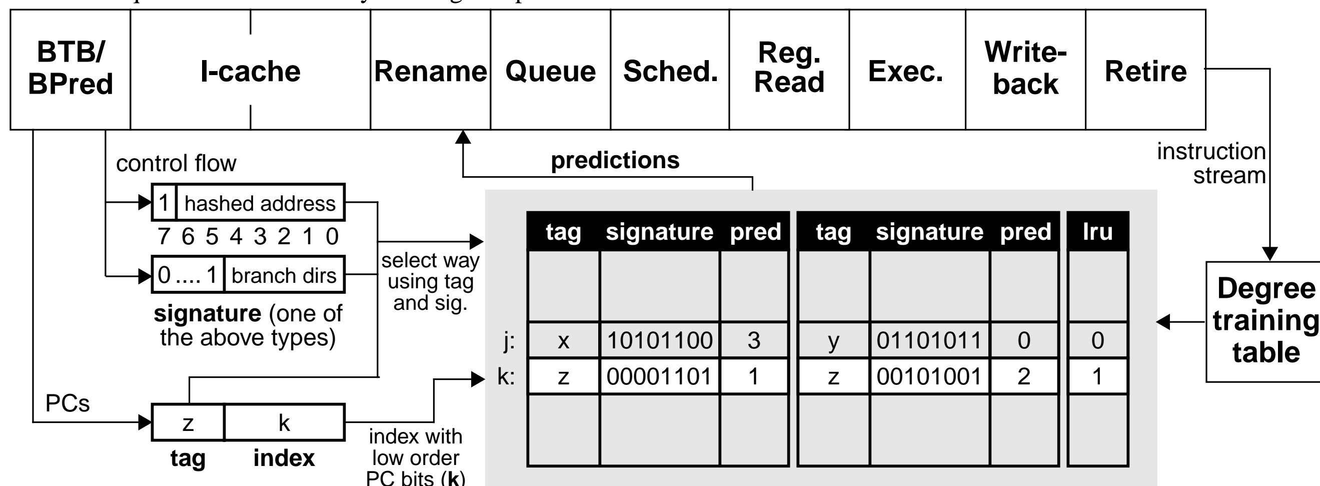


Figure 7. Degree of use predictor organization. The set-associative predictor table is indexed with the low-order PC word-address bits of the value-generating instruction (**k** in the figure) as soon as the PC is available. Each entry contains a tag (a subset of the higher order PC bits, **z** in the figure), a signature, and a predicted degree of use. The signature encodes either the target address of a subsequent indirect jump or the number and direction of branches occurring *after* the instruction that will lead to the stored degree of use being observed. The current signature is fed forward from the front end to select the correct prediction in the rename stage. Both the tag and signature must match to generate a prediction. The degree training table monitors the retirement instruction stream to train the predictor.

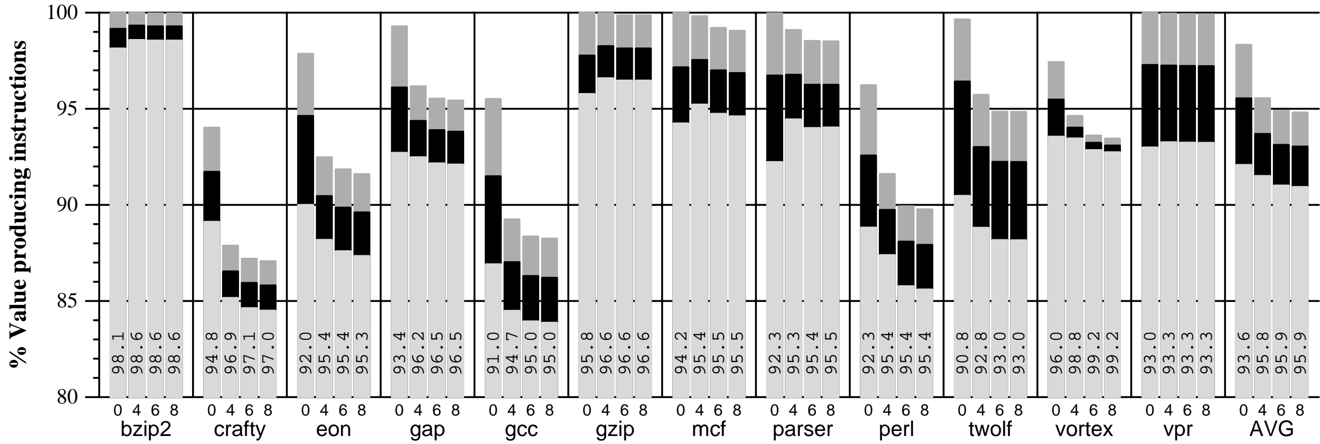


Figure 8. Degree of use predictor performance as a function of signature length. The number of bits in the control flow signature is shown on the x-axis. The stacked bars show the results of each prediction (correct, underpredicted, and overpredicted from bottom to top) normalized to the number of value-generating instructions. The numerical accuracy is printed on each bar while coverage may be read by the height of the stacked bars. All predictors have 4k entries (1k-sets by 4-way set-associative). Note the offset of the y-axis.

predictions that are correct. We note that increasing coverage alone increases correct predictions *and* mispredictions, while increasing accuracy alone changes mispredictions into correct predictions. Provided the accuracy is above a minimum threshold (established by the benefit of correct predictions versus the cost of mispredictions), increasing either accuracy or coverage in isolation results in a net benefit. Many of the predictor modifications that we propose involve a trade-off between accuracy and coverage. Whether such a trade-off is beneficial ultimately depends on the application.

Figure 8 shows the performance of the predictor as a function of the length of the control flow signature. The leftmost configuration (the "0" bar in the figure) for each benchmark shows the performance of a basic cache-like predictor (*i.e.*, one without a control flow signature) for comparison. Adding signature bits makes the predictor more discriminating but reduces the number of predictions made. Thus, accuracy increases while coverage is reduced. Coverage is impacted the most for those benchmarks with highly variable control flow (*e.g.*, `crafty` and `gcc`). The gain in accuracy is determined by the predictability of the control flow and the ability of the available control flow to fix the degree of use.

While the predictor without a control flow signature performs well for most benchmarks, an average accuracy of almost 96% is available with only four to six signature bits. Beyond six bits, the performance is relatively insensitive to the signature length since the pipeline depth limits the number of branch predictions available for inclusion in the signature. Referring back to Figure 7, we note that the number of branch directions that can be encoded in a control flow signature is two less than the number of signature bits. Thus, a six-bit signature can hold up to four predicted branch directions. Longer signatures primarily improve the predictor's ability to distinguish indirect branch targets. We use six signature bits unless otherwise noted for subsequent results.

Two fundamental variables in the design of a cache-like predictor are its size and associativity. These variables are the primary determinants of predictor coverage. Figure 9 shows how the coverage of the predictor depends on the predictor organization. Unsurprisingly, higher capacity yields higher coverage. As is true for other kinds of caches, increasing associativity also provides a benefit although the majority of the benchmarks benefit very little from associativities beyond four. The knee of the curve occurs at about 4K-entries for the associative predictors. Therefore, we use a 1K-set by 4-way set-associative predictor as our base case for the remainder of the paper. The effect of predictor organization on accuracy is negligible: the accuracy of every configuration shown in Figure 9 was within 0.5%. Capacity only affects the availability of a prediction, not its accuracy.

To this point, all of the results we have presented have assumed enough tag bits to perfectly distinguish between all static instructions in our benchmark set. These tag bits

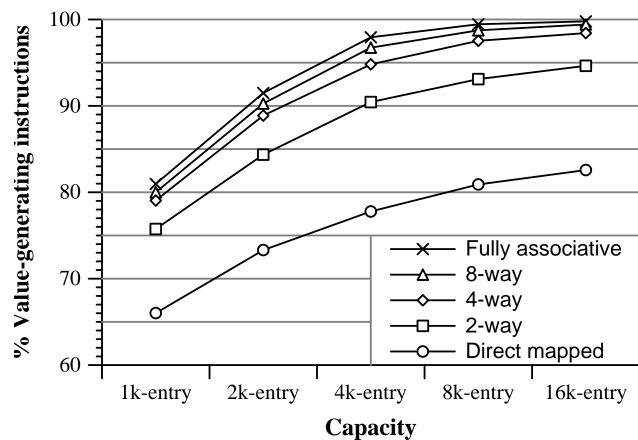


Figure 9. Predictor coverage. This figure shows how coverage of the predictor depends on the its organization. Each data point represents the average coverage over the twelve benchmarks. Note the offset of the y-axis.

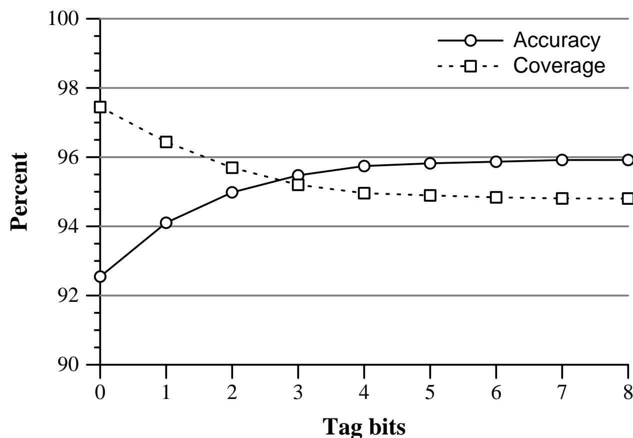


Figure 10. Effect of tag length on predictor performance. All configurations employ a 10-bit set address (*i.e.*, the predictor with four tag bits uses the low-order 14 instruction word-address bits to distinguish entries). Each data point represents the average over all the benchmarks. Note the offset of the y-axis.

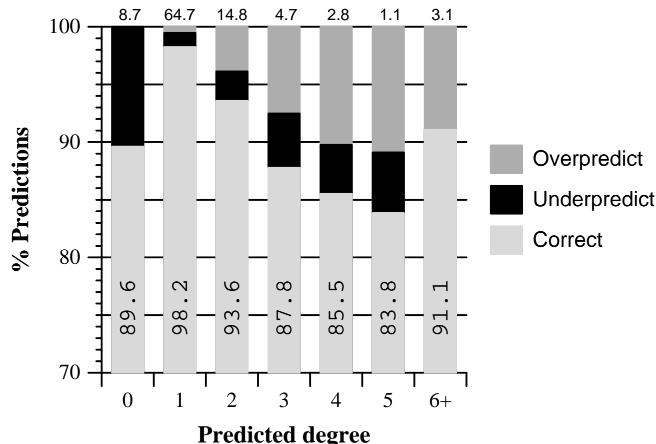


Figure 11. Predictor performance by predicted degree. Outcomes of predictions for each predicted degree of use. The numerical per-degree accuracy is printed on each bar while the percentage of all predictions accounted for by each degree appears above each bar. Note the offset of the y-axis.

come from the higher order bits of the target instruction PC, while the lower order bits are used in indexing the predictor. Reducing the number of tag bits can lead to aliasing as different static instructions become indistinguishable. While the probability of destructive aliasing is higher than in a branch predictor, for example, due to the high number of possible prediction outcomes, the occurrence of aliasing is lessened by the requirement for a control flow signature match. Figure 10 shows that aliasing is not a significant problem in a degree of use predictor. Together with the ten bits used to select the predictor set, six additional tag bits are sufficient to obtain nearly all of the benefit of tagging.

While the data in Figure 8 showed aggregate performance of the predictor, Figure 11 breaks down the predictor performance by predicted degree, revealing several interesting features. Most striking is the very high accuracy of predictions of degree of use one. This is attributable to the fact that most instructions generating a degree of use of one communicate with a single unique consumer throughout the execution of the program. Although degree of use one predictions are by far the most accurate, they account for more total mispredictions than any other degree (noted by scaling the accuracies in Figure 11 by the relative numbers of predictions above the bars). This is due to the large number of degree of use one predictions generated (65% of all predictions) relative to predictions of other degrees. Degree of use zero values are difficult to predict compared to other low degrees of use. Another interesting trend is the degradation of accuracy with increasing degree of use. The lifetime of values with higher degrees of use rapidly exceeds the depth of the pipeline from the generation of the value to the front end. The consequence is that control flow information that influences the number of uses is beyond the reach of the signature, lowering the prediction accuracy. The higher accuracy of the limiting degree category is inflated by the inclusion of multiple degrees of use and the absence of underpredictions.

5.4 Fine tuning the degree of use predictor

In this section, we explore changes to predictor algorithms that result in improved accuracy or coverage or lower overhead. The performance of all of the alternative algorithms is portrayed in Figure 12 along with the base case performance for easy comparison. Due to space limitations, we omitted benchmarks that performed well in the base case or were not strongly affected by the changes proposed in this section. The averages shown on the right side of Figure 12 include the omitted benchmarks, however.

One optimization that could improve predictor accuracy is the addition of a two-bit saturating confidence counter per entry. The same mechanism benefited the dead instruction predictor of our prior work [2]. Each counter can take on values between zero and three, inclusive. New entries are initialized with a confidence of two and a prediction is only made if the confidence is two or greater. The confidence is modified during training when there is a write hit. A hit with the same degree of use increases the confidence while a hit with a different degree of use decreases it. If the confidence decreases to zero, the stored degree is replaced. Comparing the C bars to the B bars in Figure 12 show that adding a confidence mechanism results in both a 40% average reduction in mispredictions *and* a larger absolute number of correct predictions. Thus, although the coverage decreases, the simultaneous accuracy increase actually leads to more benefit.

When introducing control flow signatures in Section 5.2, we referred to the possibility of using alternate signature handling policies. One such change is to relax the signature matching rule for branch direction based signatures. Instead of requiring that both signa-

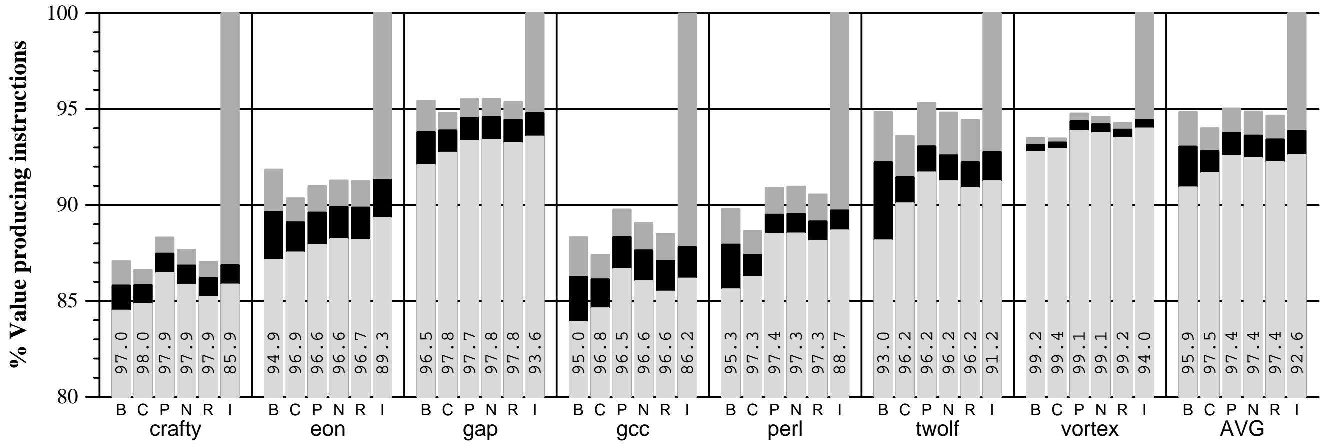


Figure 12. Evaluation of enhancements to a degree of use predictor. Results of each prediction (correct, underpredicted, and overpredicted from bottom to top) normalized to the number of value-generating instructions for a selected subset of the benchmarks. Average is over all twelve benchmarks. The numerical accuracy is printed on each bar while coverage may be read by the height of the stacked bars. All predictors have 1k-sets and are 4-way set-associative with 6-bit tags and signatures. The configurations are (B) baseline; (C) B + 2-bit confidence counter; (P) C + prefix matching; (N) P + not-MRU replacement; (R) P + random replacement; and (I) R + implicit prediction of highest degree class. Note the offset of the y-axis.

tures match exactly, we can allow a match when the stored signature is a prefix of the current signature. In these cases, the current control flow has satisfied the requirement imposed by the stored signature, so it should be able to use the stored prediction. The data show that on average, this is a better policy than requiring an exact match (P vs. C). This time, the accuracy is degraded slightly, but the increase in coverage results in more total correct predictions with a negligible increase in mispredictions.

While the previous optimizations improved predictor performance, we now examine the performance cost of reducing the storage overhead of the predictor. The use of a set-associative structure requires the implementation of a replacement policy. Up to this point, we have used a perfect LRU strategy. Other replacement possibilities with lower overhead are not-MRU and random. The performance of these alternatives is shown in the N and R bars of Figure 12, respectively. Compared with the perfect LRU scheme (P bars), using a simpler replacement policy only slightly impacts the predictor coverage. This loss results from the premature eviction of useful entries.

Another potential means of increasing predictor coverage is to assume a default prediction. In a tagged predictor, there are two possible outcomes when a prediction is requested. If the predictor has knowledge of the instruction (*i.e.*, a "hit" in the predictor), the predicted degree is returned. On a miss, however, the predictor has the option of returning a default prediction. Looking at the data presented in Figure 2, it seems that the ideal default prediction would be to predict a degree of use of one, since it is the most common degree of use (and remains so among non-predicted values). However, we will argue in Section 5.5 that degree overpredictions are less costly than degree underpredictions. Under this assumption, we can interpret a predictor miss as an implicit prediction of the highest possible degree of use. This allows us to avoid explicitly storing entries for degrees in this class, increasing the number of predictor entries available to distinguish among the lower degrees of use. The I bars of Figure 12 show these results (compare to the R bars). While the number of underpredictions is unchanged, we increase the number of correct predictions at the cost of a large number of overpredictions. Depending on the particular predictor application, an overprediction may not result in a performance penalty and thus this optimization would be worth considering. We revisit this issue in Section 5.5.

Our final tuned predictor is represented by the R configuration of Figure 12. This is a 4K-entry, 4-way set-associative degree predictor with random replacement. Each entry is augmented with a two-bit saturating confidence counter and a six-bit forward control flow signature. The stored signature is assumed to match any signature of which it is a prefix. The hardware requirements of such a predictor are modest. Each of the 4K entries consists of a three-bit degree of use, a six-bit signature, a six-bit tag, and a two-bit confidence, resulting in a storage requirement of only 8.5 KB. The contents of each predictor entry are diagrammed in Figure 13. We use this predictor in the next section to explore the issues involved in determining the outcome of a degree of use prediction.

5.5 Verifying degree of use predictions

Degree of use prediction is unique in that prediction verification depends not on the instruction for which the prediction was generated, but on some other instruction sharing a destination register name. The consequence of this is that process of verifying a degree of use prediction is involved (see Section 5.1) and the time required can be very long and depends on both the program characteris-

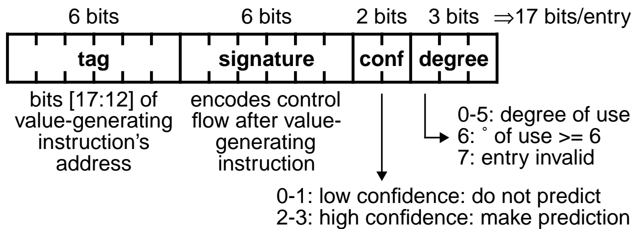


Figure 13. Degree of use predictor entry. Contents and field widths of each entry in our model predictor.

tics and the processor. The number of intervening instructions between consecutive definitions of the same register is a property of the program; the time taken to process these instruction, however, depends on the details of the microarchitecture. The verification latency ultimately determines the amount of speculative state that must be maintained to apply degree of use predictions. Long verification latencies lead to long misprediction recovery times, increasing the cost of mispredictions.

The median time to determine the outcome of a prediction is about 36 cycles. Figure 14 breaks down this verification latency by predicted degree and prediction outcome. For most benchmarks, verification latencies increase with predicted degree (the aggregate results of Figure 14 are skewed by the `mcf` benchmark; due to cache misses, it exhibits very long verification latencies for the lower degrees of use). This is to be expected since a higher degree of use requires more instructions between subsequent definitions of the value. The time to detect an overprediction closely tracks the time to verify a correct prediction since both require the retirement of the instruction overwriting the value. Detecting an underprediction requires only the retirement of one more use than was predicted; thus, this latency deviates further from the verification latency and is lower for the most common degrees of use (zero through two).

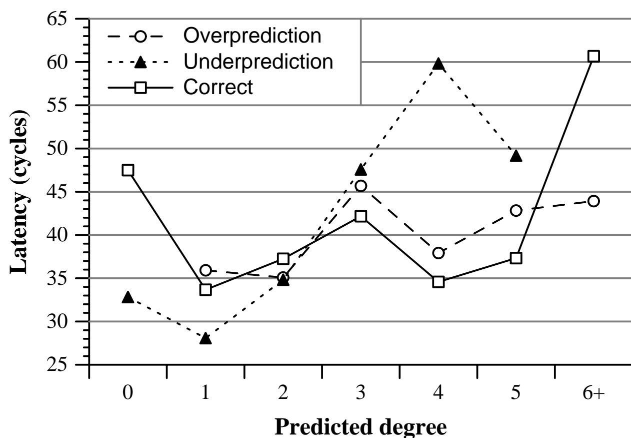


Figure 14. Prediction verification latency. Plot of the median number of cycles to determine the outcome of a prediction versus the predicted degree and outcome. Results are averages over the twelve benchmarks. Note the offset of the y-axis.

Referring back to the pipeline diagram in Figure 7, we see that a minimum of six cycles is required to verify a prediction. This corresponds to an instruction and its verifying instruction (the one that overwrites the same register) being renamed in the same cycle and traveling through to the retirement stage in the minimum possible time. Any contention or data cache misses that occur between rename and retirement will increase this latency.

We can reduce the delays due to events in the back end of the processor pipeline by performing misprediction detection earlier in the processor pipeline. A structure similar to the DTT can perform early detection of mispredictions as soon as the predicted degree of use is available (*e.g.*, at the end of the rename stage). We call this structure a *use tracking table* or UTT. Like the DTT, the UTT also contains counters representing the number of observed uses per architectural register. Additional counters store the predicted degrees of use (if any) for the register values. If the observed use counter exceeds the predicted degree of use, a mispredict may be signalled and recovery initiated. Similarly, for early overprediction detection, a misprediction occurs when the value is overwritten prior to the predicted number of uses occurring.

Due to wrong path execution, false mispredictions can be signalled when spurious uses along the wrong path are observed. There exists a trade-off between how quickly a misprediction is detected and the number of false mispredictions. One can tailor the UTT to only signal mispredictions on those predictions that may require costly recovery. Given reasonably accurate branch prediction, however, the benefit of detecting degree of use mis-speculation early will likely outweigh the costs of the small number of false mispredictions. We also note that UTT contents must be recovered after any wrong path execution to avoid incorrect use counts.

Figure 15 shows the effect of implementing early misprediction detection on the median misprediction

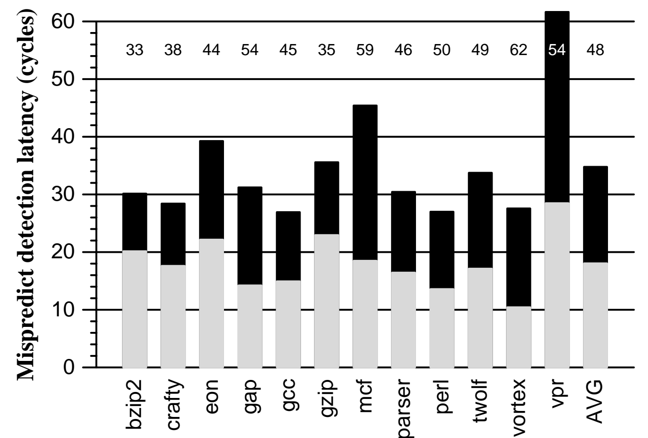


Figure 15. Early misprediction detection. Median number of cycles required to detect a misprediction. The top bar corresponds to misprediction detection at retirement while the bottom bar shows the results when using early misprediction detection. The percentage reduction is shown numerically.

detection latency. Most of the benchmarks see at least a 40% reduction in the median detection time when using early misprediction detection, and the average reduction is almost 50%. For all but one of the benchmarks the number of false mispredictions was less than 1% of the total number of predictions (for `twolf`, the figure was 1.2%). Given the high baseline prediction accuracy, however, the false mispredictions amounted to an 18% increase in total mispredictions on average (31% maximum for `twolf`).

Another important issue relating to misprediction detection is that of overpredictions versus underpredictions. We hinted in Section 5.4 that overpredictions might not be as costly as underpredictions. In general, a degree overprediction represents only a lost opportunity for optimization, not an event requiring recovery. It is difficult to imagine an application that would fail when fewer than the predicted number of uses occurs; the same is not true when the number of uses is underpredicted. In these cases, the value may have to be recovered from a special structure or altogether regenerated for uses beyond the number predicted. Because of this distinction, the predictor optimization involving an implicit maximum prediction would probably benefit most, if not all, applications of a degree of use predictor even though the number of overpredictions is greatly increased.

The issue of the highest predictable degree is also germane. Optimizations involving degree of use prediction will leverage the most common and accurately predictable degrees of use, one and two. Given the small contribution of the higher degrees of use and the lower accuracy of their prediction (Figure 9), we expect that most degree of use predictors will not attempt to differentiate degrees of use greater than two or three. Beneficial side effects of reducing the prediction limit are the elimination of underpredictions involving degrees greater than the limit, resulting in increased predictor accuracy, and a potential decrease in the size of the predictor (if the degree field can be made smaller).

6 Conclusions and future work

Learning about the communication patterns of register values is the first step in streamlining inter-instruction communication in processor cores. Observing that a value's degree of use is the primary determinant of a value's communication requirements, we assert that a degree of use predictor will be a key component of many communication optimizations. To this end, we studied a variety of accurate degree of use predictors. One particular predictor provides correct predictions for better than 92% of all dynamic values using a small amount of hardware, results that are not atypical for other variations.

The predictors we describe leverage two key properties of dynamic degree of use that we studied in this paper. First, degree of use exhibits locality. Most static instructions generate values with the same degree of use during every execution. When an instruction can produce values with multiple degrees of use, values produced consecutively are still likely to have the same degree of use. Second, the control flow occurring soon after a value is generated often perfectly establishes that value's degree of use. Due to the pipelined nature of processors, we can successfully employ control flow predictions available for these near future instructions in generating predictions.

The proof of accurate degree of use prediction is a prerequisite for each of the optimizations we suggested in the beginning of the paper. Although we do not evaluate a specific application in this paper, we have already described one application for utilizing the degree of use zero predictions [2]. We are actively exploring inter-instruction communication methods that exploit the preponderance of values with degree of use one.

Acknowledgements

The authors would like to thank Brian Fields, Paramjit Oberoi, Erik Paulson, Amir Roth, Brandon Schwartz, and Craig Zilles for their helpful comments and assistance.

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] J. Adam Butts and G. Sohi. Dynamic dead-instruction detection and elimination. In *Proc. of the 10th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, October 2002. pp. 199-211.

[3] J. Cruz, A. Gonzalez, M. Valero, and N. Topham. Multiple-banked register file architectures. In *Proc. of the 27th Annual Int'l Symp. on Computer Architecture*, June 2000. pp. 316-25.

[4] L. Eeckhout and K. Bosschere. Hybrid analytical-statistical modeling for efficiently exploring architecture and workload design spaces. In *Proc. of the 2001 Int'l Conf. on Parallel Architectures and Compilation Techniques*, September 2001. pp. 25-34.

[5] M. Franklin and G. Sohi. Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors. In *Proc. of the 25th Int'l Symp. on Microarchitecture*, December 1992. pp. 236-45.

[6] H. Kim and J. Smith. An instruction set and microarchitecture for instruction level distributed processing. In *Proc. of the 29th Annual Int'l Symp. on Computer Architecture*, May 2002. pp. 71-81.

[7] S. Palacharla, N. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proc. of the 24th Int'l Symp. on Computer Architecture*, June 1997. pp. 206-18.

[8] J. Philips and S. Vassiliadis. High performance 3-1 interlock collapsing ALUs. In *IEEE Trans. on Computers*, vol. 43, March 1994. pp. 257-68.