Improving Context-Based Load Value Prediction

by

Martin Burtscher

Dipl.-Ing., Eidgenössische Technische Hochschule, 1996

A thesis submitted to the
Faculty of the Graduate School of the
University of Colorado in partial fulfillment
of the requirement for the degree of
Doctor of Philosophy
Department of Computer Science
2000

This thesis entitled:
Improving Context-Based Load Value Prediction
written by Martin Burtscher
has been approved for the Department of Computer Science

Benjamin G. Zorn

Michael Franz

Dirk Grunwald

William M. Waite

James H. Martin

Date

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Burtscher, Martin (Ph.D., Computer Science)
Improving Context-Based Load Value Prediction
Thesis directed by Associate Professor Benjamin G. Zorn

Abstract

Microprocessors are becoming faster at such a rapid pace that other components like random access memory cannot keep up. As a result, the latency of load instructions grows constantly and already often impedes processor performance.

Fortunately, load instructions frequently fetch predictable sequences of values. Load value predictors exploit this behavior to predict the results of load instructions. Because the predicted values are available before the memory can deliver the true load values, the CPU is able to speculatively continue processing without having to wait for memory accesses to complete, which improves the execution speed.

The contributions of this dissertation to the area of load value prediction include a novel technique to decrease the number of mispredictions, a predictor design that increases the hardware utilization and thus the number of correctly predicted load values, a detailed analysis of hybrid predictor combinations to determine components that complement each other well, and several approaches to substantially reduce the size of hybrid load value predictors without affecting their performance.

One result of this research is a very small yet high-performing load value predictor. Cycle-accurate simulations of a four-way superscalar microprocessor running SPECint95 show that this predictor outperforms other predictors from the literature by twenty or more percent over a wide range of sizes. With about fifteen kilobytes of state, the smallest examined configuration, it surpasses the speedups delivered by other, five-times larger predictors both with a re-fetch and a re-execute misprediction recovery mechanism.

Dedication

To my family.

Acknowledgements

Many wonderful people have helped me succeed in academics and life. First and foremost, I would like to thank my mother, whose never-ending encouragement, patience, and help doubtlessly brought me to where I am today, and my father, who motivated and supported my interest in computers and sciences a great deal.

Many teachers have made lasting positive impressions on me and have provided invaluable guidance. Most importantly, it has been a great experience and pleasure to work with my advisor Professor Benjamin Zorn, whose help, support, and insights I am extremely grateful for. He always knew how and when to guide me while still allowing me immense freedom and flexibility with my work, which I appreciate very much. But also Professors Michael Franz, Dirk Grunwald, Amer Diwan, William Waite, and James Martin, as well as my fellow students and many other great people have shaped my career and success to no small amount.

I would like to especially thank my fiancée Anna Szczyrba for her encouragement and support of my work and for bearing with me all this time.

This work was funded in part by the Hewlett Packard University Grants Program (including Gift No. 31041.1) and the Colorado Advanced Software Institute. I would like to thank Tom Christian for his support of this project and Dirk Grunwald and Abhijit Paithankar for providing and helping with the cycle-accurate simulator. In addition to using Hewlett Packard desktop computers, most of the simulations were performed on Alpha machines, which were sponsored by a Digital Equipment Corporation (now Compaq) grant.

Contents

Tables

Figures

Chapter 1

Introduction

This chapter describes how the slow execution speed (the latency) of load instructions can impact the performance of a processor and introduces load value prediction, a promising approach to alleviate the load latency problem.

Furthermore, the contributions of this dissertation to the area of load value prediction are presented.

1.1 Problem

Processor technology is advancing at a rapid pace. Over the past two decades the CPU speed has roughly doubled every one and a half years (this is informally known as Moore's Law). To continue this trend and to satisfy the incessantly growing need for more computing power, novel techniques are needed to make microprocessors faster and faster. This dissertation explores and improves one such technique called *load value prediction*.

While the CPU performance has been accelerating at a high speed, the advances in other areas (such as the reduction of the memory latency) have not been as dramatic. As a consequence, memory accesses have in relative terms become slower over the years and have reached a point where they present one of the biggest processor performance bottlenecks. Load value prediction reduces the effective memory latency and thus speeds up the CPU.

Load instructions copy data from memory to a register inside the CPU. The register that receives the data is called the *target register* and is specified

in the load instruction. Registers can be accessed very rapidly, but a CPU can only have relatively few of them (usually fewer than about sixty-four). The memory, on the other hand, can hold over a million times more data than the register file but accessing it takes on the order of a hundred times longer, making such accesses very time consuming.

To reduce the access time of frequently used data, most computers incorporate levels of fast cache memory. The first cache level (L1) is normally the smallest but also the fastest and temporarily stores the most recently used data in case it is needed again. Consecutive levels are larger and slower. The main memory is at the end of the (volatile) memory hierarchy and has the longest access time. When a load instruction is executed, the caches are successively queried until the desired data item is found. If the L1 cache contains that data, the load value will be available quickly. If the data cannot be found in any cache, the data has to be retrieved from the main memory. Hence, the time it takes to execute a load instruction depends on the cache level that satisfies the load request and can vary from a few cycles to over a hundred cycles. In comparison, reading a value from the CPU's register file never takes longer than one cycle.

Because the technological enhancements have improved CPUs more than memory chips, the speed-gap between CPU and memory grows constantly, making load instructions slower and slower relative to the CPU. If this trend continues, and there is currently no indication that it will not, the load latency will become even longer and more of a problem in the future.

Load instructions belong to the most frequently executed instructions. Many programs, even highly optimized ones, execute more than one load for every five executed instructions [LCB+98]. Hence, the latency of load instructions can, and frequently does, hamper system performance. Conversely, reducing the (effective) load latency has the potential to substantially speed up program execution.

Only branch instructions present a similarly substantial source of over-

head in modern microprocessors. While extensive research has been performed to alleviate the performance impact of branches (for example by using sophisticated branch predictors [LCM97], branch target buffers [PeSm93], and return address stacks [KaEm91]), relatively little has been done to address the load latency problem.

As opposed to load instructions, latency is not an issue with store instructions because their (slow) memory access takes place “after” the execution of the store, i.e., the CPU can proceed without having to wait for the store to complete. Write buffers [Jou93] perform the actual store operation at some later time and make sure that consistency is maintained.

Unfortunately, nothing similar can be done for load instructions because the fetched values are often almost instantly needed by the immediately following instructions. These instructions cannot execute before the load they depend on has completed. Even worse, all the indirectly dependent instructions are also delayed until the load has completed.

1.2 Load Value Locality

Fortunately, load instructions often fetch predictable sequences of values [LWS96]. For instance, about half of all the load instructions in the SPECint95 benchmark suite retrieve the same value that they did the previous time they were executed. Such behavior, which has been demonstrated explicitly on a number of architectures, is referred to as *value locality* [Gab96, LWS96]. The predictability of load values can be exploited by predicting the result of a load instruction before the memory can provide the load value.

Several distinct types of load value locality have been identified so far and predictors to exploit them have been proposed [BuZo99b, Gab96, LWS96, SaSm97b, TuSe99, WaFr97]. The main goal of this dissertation is to develop and evaluate new and better performing load value predictors.

If load values are predicted quickly and correctly, the CPU is able to con-

tinue processing the dependent instructions without having to wait for the memory access to finish. Of course it is only known whether a prediction was correct once the true value has been retrieved from memory, which can take many cycles. *Speculative execution* allows the CPU to continue execution with a predicted value before the prediction outcome is known [John91]. If it later turns out that the prediction was correct, the speculative status can simply be dropped. If the prediction was incorrect, everything that the CPU did using the incorrect value has to be purged and redone with the correct value.

Because branch predictors require a similar mechanism to recover from mispredictions, most modern CPUs already contain the necessary hardware to perform this kind of speculation [Gab96]. However, recovering from mispredictions takes time and slows down the processor. Load value prediction therefore only makes sense if the predictions are often correct. Improving the accuracy of load value predictions is another goal of this thesis.

Empirically, papers have shown that the results of most instructions are predictable [Gab96, LiSh96, SaSm97a]. While predicting the result of every instruction potentially enables wide issue CPUs to exceed the existing instruction level parallelism (ILP) [GaMe98, LiSh96], predicting only load values requires substantially less and simpler hardware while still yielding most of the performance potential found in value prediction [ReCa98], and can even be advantageous in single-issue CPUs.

1.3 Prediction Approaches

There are three basic ways to find predictable load instructions and to determine their load values. The first possibility is static prediction. This approach makes all decisions prior to program execution. Hence, the only information available is the binary, predefined heuristics, and possibly the source code. Profile-based approaches represent another possibility. They measure and record the behavior of programs for several sample inputs. Fi-

nally, dynamic approaches continuously measure the behavior of programs while they are executing.

The static approach is rather limited due to the large number of runtime constants whose values are not known at compile time [CFE97]. Profiling often suffers from insufficient coverage, i.e., not all parts of a program are executed during the profile run, which means that no information for those parts is gathered. Furthermore, both static and profile-based approaches need support in the instruction set architecture (ISA) to communicate information to the hardware. Such support is generally not available in existing CPU families. The dynamic approach does not suffer from these problems, but it requires a predictor to be present in hardware. Furthermore, the dynamic approach does not know a priori which load instructions are predictable, meaning that space has to be provided in the predictor for both predictable and unpredictable loads. Hence, it may be advantageous to combine a static or profile-based approach with a dynamic predictor to filter out the unpredictable loads so that the predictor only has to be designed large enough to handle the predictable loads [GaMe97].

Another important advantage of the dynamic approach is that it can adapt to changes in the program behavior during the course of the execution. The information provided by static approaches or by profiles is normally fixed and cannot be changed at runtime.

Due to the limitations of the static and the profile-based approaches, I will restrict my investigation to dynamic, hardware-based load value predictors that are completely transparent (i.e., do not require changes to the ISA) and can therefore be added to existing as well as future microprocessors. No profiling or compiler support is needed for my predictors.

1.4 Confidence Estimation

Thirty to fifty percent of the executed load instructions cannot be correctly

predicted with the currently known prediction techniques. Trying to predict these loads will inevitably result in mispredictions. Because recovering from mispredictions takes time, a high misprediction-rate can incur a recovery cost that eradicates any benefit that was gained from the correct predictions. Hence, it is possible for a load value predictor to slow down the processor instead of speeding it up.

To keep the number of mispredictions at a minimum, almost all load value predictors incorporate some form of *confidence estimator* that tries to identify predictions that are likely to be incorrect so that they can be inhibited. Inhibiting such predictions reduces the number of mispredictions (and the associated recovery cost) and thus improves the predictor's performance.

This dissertation presents a new confidence estimator that makes fewer mispredictions than the conventional confidence estimator and therefore results in more effective load value predictors.

1.5 Contributions

The goal of this dissertation is to develop and evaluate methods for context-based load value prediction, that is, to enhance various aspects of transparent, hardware-based load value predictors. My contributions towards this goal include the following:

- Fewer mispredictions

  The development of an improved confidence estimator that decreases the number of mispredictions and consequently increases the performance of load value predictors
- Better hardware utilization

  The design of a load value predictor that allocates more hardware to the frequently executed loads, which improves the predictor utilization and results in more load instructions being correctly predicted

- Hybrid analysis

  The analysis of a large number of hybrid predictor combinations to determine components that complement each other well and thus yield high-performing hybrid load value predictors

- Size reduction techniques

  Several approaches to substantially reduce the size of hybrid load value predictors by sharing large amounts of state between their components, which decreases the predictor size while maintaining the performance

One direct result of this research is a high-performing load value predictor that includes all of the above mentioned enhancements. It is a hybrid of well-complementing components that is very small due to the large degree of state sharing. With about fifteen kilobytes of state, it outperforms five-times larger predictors from the literature. Among predictors of similar size, my predictor outperforms others by twenty or more percent over a large range of predictor sizes. The individual contributions are discussed in a little more detail in the following paragraphs.

Analyzing the performance of an existing confidence estimator revealed a weakness that prevents it from correctly handling sequences of alternating predictability, which represent an important subset of the predictable load value sequences. To alleviate this problem, I developed a more complex confidence estimator that is somewhat larger but yields on average ten percent more performance in connection with most load value predictors. Moreover, there is evidence that the new confidence estimator embodies a better selector for hybrid load value predictors, improving the performance even further over the conventional confidence estimator.

An investigation of the utilization of the hardware in a basic load value predictor revealed that most parts of the predictor are hardly ever or never used while a small part is used extremely frequently. To improve the utilization, I studied possible rearrangements of the predictor's hardware. I found

an arrangement that allocates more hardware to the frequently executed load instructions and that is therefore able to correctly predict a larger number of loads, which improves the average predictor performance by about ten percent.

Then I noticed that many of the values this improved predictor retains differ only by a small amount. This allowed me to devise a predictor in which the values are stored in a compressed format. Compressing the values reduces the predictor size by about one half while essentially maintaining the predictor's performance.

Next I discovered that components of hybrid predictors frequently store the same information. Hence, the redundant information can be eliminated, which can reduce the size of hybrids by more than a factor of two without compromising the predictor's performance.

A detailed component analysis of a large number of predictor combinations revealed some unexpected results. For example, powerful individual components frequently do not complement each other well in a hybrid configuration. Conversely, some components that perform rather poorly when used in isolation can form strong coalitions with other components. The results of this analysis allowed me to design a hybrid out of components that are small yet complement each other well. Many other hybrid predictors were found to contain components that predict highly overlapping sets of load instructions and therefore do not ideally complement one another. Furthermore, some hybrids actually yield a lower performance than their individual components due to negative interference.

Finally, this dissertation presents performance numbers for a large number of load value predictors that are all evaluated in the same environment (i.e., the same simulator, the same benchmark programs, etc.), making it possible to truly compare the predictors. Furthermore, various performance metrics are introduced and studied, and a simple predictor banking scheme is evaluated.

1.6 Summary

One of the largest performance bottlenecks in current microprocessors is the growing load latency. Load value prediction has the potential to substantially reduce the load latency.

The main contribution of this dissertation is the development and evaluation of a high-performing yet relatively small load value predictor that significantly outperforms other predictors from the literature.

1.7 Organization

The remainder of this dissertation is organized as follows. Chapter 2 explains the impact of the load latency on modern superscalar CPUs as well as the operation of two misprediction recovery mechanisms. Chapter 3 describes the configuration of the simulator that is used to measure the speedup numbers and discusses the benchmarks and their load value locality. Chapter 4 introduces the architecture of five basic load value predictors. Chapter 5 investigates two confidence estimation schemes. Chapter 6 analyzes the performance of predictor banking. Chapter 7 takes a closer look at the utilization of the predictor hardware and proposes an improved design. Chapter 8 evaluates a large number of predictor combinations to build well performing hybrids. Chapter 9 improves the results from Chapter 8 by investigating ways to shrink the size of predictors through hardware reuse. Chapter 10 presents related work. Chapter 11 summarizes my work and takes a look into the future.

Chapter 2

Background

This chapter provides background on several features of high-performance microprocessors, including parallel instruction execution (superscalar execution), the dynamic re-sequencing of the execution order of independent instructions (out-of-order execution), and their interaction with load value predictors.

Two misprediction recovery mechanisms are presented. The first one, which is the one that is also used for recovering from branch mispredictions, is already implemented in current processors but does not yield the best performance in combination with load value predictors. This is why a better, not yet implemented alternative is also discussed.

2.1 Conventional High-Performance Processor Architecture

Most of today's high-performance microprocessors are superscalar and have built-in hardware support for speculative and out-of-order execution [Edm+95, Half95, Yeag96, You94]. Since it is my goal to improve the performance of a high-end microprocessor, all the performance numbers presented in this dissertation are based on such a CPU. The specifications of the actual processor that is used for these measurements are described in Section 3.1.

A superscalar CPU is capable of executing more than one instruction at a time. Out-of-order execution refers to the ability to dynamically adjust the order in which instructions are executed to increase the utilization of the avail-

able hardware and thus to improve the performance. Speculative execution is execution that can be undone if necessary. Having this ability makes it possible to process instructions whose execution and/or input values are based on a guess (such as a predicted branch outcome or a predicted load value) because it may later be necessary to undo the execution of such instructions if the guess turns out to be incorrect.

Only the execution-core of a processor usually handles instructions out-of-order. Instruction fetch, decode, rename, and retirement is performed in-order [SmSo95] because dependencies are either not known in these pipeline stages or need to be handled in-order to facilitate correct execution. Instructions are retired in-order to support precise exceptions, to be able to replay instructions, and to enforce sequential semantics.

Register renaming removes false dependencies from the in-flight instructions by dynamically mapping the logical registers to a larger set of physical registers, thus ensuring that instructions that have their input operands available are truly independent and can therefore be executed in any order or even in parallel with all other ready instructions.

The renamed instructions are (at least conceptually) fed into the CPU's instruction window as long as there are slots available. An inserted instruction remains in this window in a waiting state until all of its source operands are available. Once all the inputs are obtained, the instruction becomes ready, i.e., eligible for execution. The CPU's issue logic continuously scans the instruction window for such instructions. If a ready instruction is found and a functional unit capable of executing that type of instruction is available, the issue logic assigns the instruction to the functional unit for execution. At this point, the instruction is marked as executing. Once the functional unit has completed the execution, the result is stored and forwarded to the waiting instructions, making them ready if the current result was the last input operand they were waiting for. Completed instructions are marked as done. Only instructions marked as done can be retired (or committed) from the instruc-

tion window.

Superscalar processors are able to locate and issue multiple ready instructions per cycle (with a fixed upper limit), as long as there are enough functional units (FUs) and ready instructions available. In addition, they are able to forward multiple results per cycle to waiting instructions.

Most of the FUs in high-performance CPUs are either pipelined (i.e., they can start executing a new instruction every cycle) or they only have a one-cycle latency. The most frequently used FUs are often duplicated for faster execution. Figure 2.1 shows the described pipeline stages, the instruction window, the issue logic, and several functional units. The instruction window contains some sample instructions in different stages of execution (i.e., waiting, ready, executing, and done). Actual CPU implementations may vary from this diagram (for example, most processors contain more slots in the instruction window than are depicted).

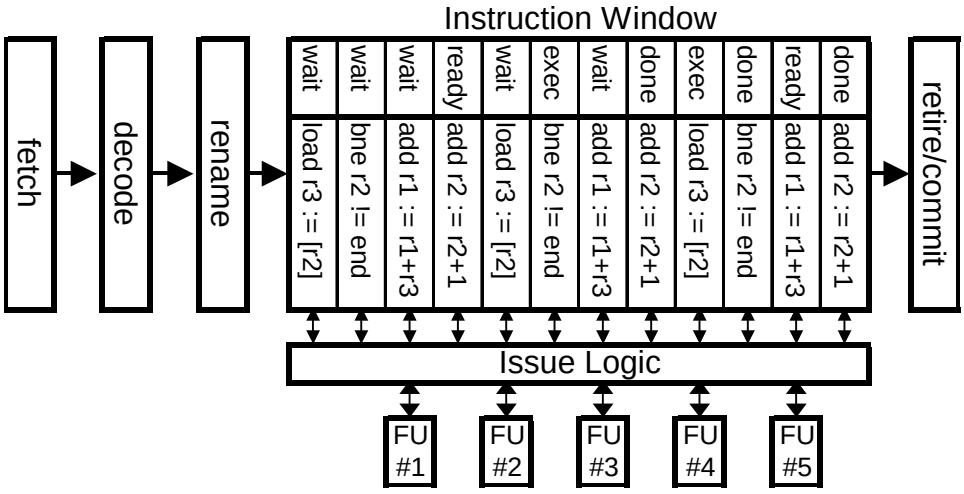

Figure 2.1: The execution pipeline of a high-performance microprocessor.

To improve their performance, some processors predict the outcome of conditional branch instructions so that they can continue fetching instructions and feeding them into the instruction window without having to wait for the

branches to resolve. All the instructions that follow a predicted branch are executed *speculatively* until the branch outcome is known. If it turns out that the branch was predicted incorrectly, the speculatively executed instructions have to be purged. Doing so is possible because all the instructions that follow the branch must stay in the instruction window at least until the branch has been resolved (because instructions are retired in-order). Whenever a prediction is made, a copy of the internal processor state is made, called a *check-point*, which is restored after a misprediction to reestablish the correct architectural state [John91]. This allows the CPU to continue with the program execution as though it had never made a misprediction. Of course, performing such recovery actions takes time (usually on the order of a few cycles), which slows down the CPU. Correct speculations, on the other hand, save cycles because some instructions are able to execute that would not have been able to if they had had to wait for the branch to resolve.

Since the speculation support necessary for value prediction is essentially identical to the one used with branch prediction [Gab96], no novel hardware is required to recover from load value mispredictions.

Modern processors are able to hide some of the occurring load latencies by executing independent instructions out-of-order. However, it is unlikely that a CPU will find enough instructions to keep itself busy for eighty cycles, which corresponds to the load-to-use memory access latency on a DEC Alpha 21264 [KMW98]. Allowing the CPU to already execute the load-dependent instructions while the memory access is still in progress potentially frees a large number of instructions in the instruction window for execution, whose advanced execution can substantially boost the performance. In fact, even during the execution of short-latency loads the issue logic may not be able to keep all the functional units busy because of a quickly vanishing selection of available ready instructions. Hence, it may be advantageous to predict short-latency loads as well.

Since all the load value predictors discussed in this dissertation require

only the load instruction's address (PC) for making a load value prediction, the prediction can be started as soon as a load instruction has been decoded. The memory access, on the other hand, cannot be initiated before the effective address has been computed, which can take several cycles. As a consequence, it is even beneficial to predict loads that hit in the L1-cache because the predicted value is available before the cache can satisfy the load request.

2.2 Misprediction Recovery Mechanisms

Two misprediction recovery mechanisms have been proposed for load value prediction. The simpler but less powerful *re-fetch* mechanism is the one already used for recovering from branch mispredictions [Gab96]. When a misprediction is detected in this scheme, all the instructions that follow a mispredicted instruction are purged from the instruction window and the processor state is reset to what it would have been had no instruction beyond the mispredicted one executed. The CPU then continues processing instructions by fetching the next instruction, that is, the instruction that immediately follows the instruction that was mispredicted. Because the purged instructions are re-fetched, I call this misprediction recovery mechanism *re-fetch*.

Re-fetch recovery incurs a cycle-penalty because it takes time to purge instructions from the instruction window and to restore the CPU's state. Even worse, in this scheme instructions are sometimes purged whose results are correct. For example, if instruction X is independent of an earlier load instruction L, then X may execute in an out-of-order processor before the load is completed. Because instruction X is independent of L, its result is also independent on the load value. Purging X is therefore not necessary, even in the presence of a mispredicted value for L.

In fact, mispredicting L does not even invalidate the instructions that do depend on L (up to the first conditional branch instruction whose branch tar-

get depends on L). In the worst case, these instructions are executed with an incorrect input value. Because all the affected instructions remain in the instruction window, it suffices to re-execute them with the correct input value [LiSh96]. Consequently, the state of the directly and indirectly dependent instructions “only” needs to be reset to ready or waiting after a misprediction so that the issue logic will select them again for execution. This second (or subsequent) execution will produce the correct result because the input operands are now correct. I refer to this misprediction recovery mechanism as *re-execute* recovery.

While the re-execute mechanism avoids the unnecessary purging of independent instructions and the overhead of re-fetching already fetched instructions, it still incurs a cycle-penalty for identifying the dependent instructions and changing their state. However, the penalty is considerably smaller than the one incurred by the re-fetch recovery mechanism. Note that, as opposed to re-fetch hardware, re-execute hardware does not yet exist and incorporating it requires changes to the CPU core, which may or may not be cost-effective.

2.3 Summary

Superscalar execution, out-of-order execution, register renaming, and branch prediction are but a few of the techniques used to improve the performance of microprocessors. Some of these features are able to hide the access latency of load instructions to a certain degree. Nevertheless, a substantial and growing load latency problem remains. Load value predictors present a promising new approach to alleviate this problem.

Because predictions are sometimes wrong, misprediction recovery mechanisms are needed. The mechanism that is used for recovering from branch mispredictions can readily be applied to load value prediction. Unfortunately, it is rather conservative and hampers performance, which is why an

alternative recovery mechanism has also been proposed, which results in better performance.

Chapter 3

Evaluation Methods

This chapter describes the configuration of the baseline CPU that is used for the cycle-accurate simulations, gives information about the benchmark programs that are used for the performance evaluations, and presents the metrics used to measure the effectiveness of load value predictors.

3.1 Baseline Architecture

All measurements in this dissertation are based on the DEC Alpha AXP architecture [DEC92]. The various load value predictor designs are evaluated using the ATOM binary instrumentation tool-kit [EuSr94, SrEu94] and the AINT simulator [Pai96] with a cycle-accurate superscalar back-end.

ATOM is used to instrument the benchmark suite (see next section) for fast and thorough parameter-space evaluations because of its speed and ease of simulating the proposed predictors in software. Promising configurations are then fed to the pipeline-level simulator for more detailed measurements.

The simulator is configured to emulate a high-performance microprocessor similar to the DEC Alpha 21264 [KMW98]. It accurately models the processor's internal timing behavior, resource constraints, and speculative execution as well as the memory hierarchy and its latencies. Only bus-contention is not modeled. Such detailed simulations are necessary to obtain realistic performance results. Unfortunately, they are about two orders of magnitude slower than ATOM simulations.

The simulated CPU is four-way superscalar, issues instructions out-of-order from a 128-entry instruction window, has a 32-entry load/store buffer, four integer and two floating point units, a 64kB two-way set associative L1 instruction-cache, a 64kB two-way set associative L1 data-cache, a 4MB unified direct-mapped L2 cache, a 4096-entry branch target buffer (BTB), and a 2048-line hybrid gshare-bimodal branch predictor. The three caches have a block size of 32 bytes. The modeled latencies are shown in Table 3.1. The six functional units are fully pipelined and each unit can execute all operations in its class. Operating system calls are executed but not simulated, which should not be a problem since the benchmark programs used for this thesis hardly perform any operating system calls [SPEC95]. Loads can only execute when all prior store addresses are known. Up to four load instructions are able to issue per cycle. This CPU represents the baseline processor (CPU_{Base}). All reported speedups are relative to CPU_{Base}, which does not contain a load value predictor.

Operation	Latency
integer multiply	8-14
conditional move	2
other int and logical	1
floating point multiply	4
floating point divide	16
other floating point	4
L1 load-to-use	1
L2 load-to-use	12
memory load-to-use	80

Table 3.1: Functional unit and memory latencies (in cycles).

In the CPUs that include a load value predictor (CPU_{LVP}), predictions take place during the rename-stage in the instruction pipeline and have a one-cycle latency. If a predictor cannot be accessed in one cycle, it has to be pipelined. Fortunately, multi-cycle access latencies can be hidden as long as there are enough stages between the decode and the execute stage in the

processor's instruction pipeline (usually two or more stages in current high-performance microprocessors).

To support up to four predictions/updates per cycle, all the load value predictors used in this study are split into four banks that can operate in parallel (see Chapter 6). Since the modeled CPU fetches naturally aligned four-tuples of instructions, it is not possible to fetch or issue two load instructions during the same cycle that go to the same predictor bank.

All the predictors are updated when the true load value becomes available (i.e., when the verification memory access completes), predictions do not speculatively update the predictor's state, out-of-date predictions are made as long as there are pending updates (for the same predictor line), and out-of-order and wrong-path updates of the predictor are accurately modeled in the simulator. All predictor updates are final and cannot be undone. Investigating the benefit of speculative updates is left for future work.

Not modeling bus-contention, assuming fully pipelined functional units, and allowing up to four load instructions to be issued per cycle reduce the average instruction latency somewhat in comparison to real CPUs. Furthermore, ignoring bus-contention also reduces the memory latency. A lower instruction latency implies more executed load instructions per time-unit, which increases the pressure on the load value predictor. Hence, the performance of a load value predictor would likely, if anything, be higher in a real CPU than the measurements in this thesis indicate due to the reduced chance of making an out-of-date prediction and the fewer dropped updates due to a busy predictor. The slightly longer-than-modeled memory latency in real systems has the same effect, i.e., it decreases the pressure on the predictor while at the same time making correct load value predictions more beneficial because of the even longer load-latency that is hidden.

Similar effects of the simulator-limitations on other parts of the CPU should cancel each other out because the baseline CPU suffers/benefits as much from them as the CPUs do that include a load value predictor.

3.2 Benchmarks

This section discusses the benchmark suite used throughout this dissertation to evaluate the performance of load value predictors.

3.2.1 General Information

All the measurements in this thesis are based on the eight integer programs of the SPEC95 benchmark suite [SPEC95]. These programs are well understood, non-synthetic, and compute-intensive, which is ideal for processor performance evaluations. Despite the lack of desktop application code, the suite is nevertheless representative thereof, as Lee et al. found [LCB+98]. The SPECint95 programs are written in C and perform the following tasks:

compress:	compresses and decompresses a file in memory
gcc:	C compiler that builds SPARC code
go:	artificial intelligence, plays the game of “GO”
ijpeg:	graphic compression and decompression
li:	Lisp interpreter
m88ksim:	Motorola 88000 chip simulator, runs a test program
perl:	manipulates strings (anagrams) and prime numbers in Perl
vortex:	an object oriented database program

The suite includes two sets of inputs for every program and allows two levels of optimization. To acquire as many load value samples as possible, the larger reference inputs are used. However, due to a restriction in the simulation infrastructure, only the first of the multiple input-files from the reference set is used with *gcc*.

To avoid possible side-effects that may be attributed to poor code quality, the peak-versions of the programs are utilized, which were compiled with

DEC GEM-CC on a DEC Alpha 21164 using the highest optimization level "*-migrate -O5 -ifo*". The optimizations include common sub-expression elimination, split lifetime analysis, code scheduling, no-op insertion, code motion and replication, loop unrolling, software pipelining, local and global inlining, inter-file optimizations, and many more. In addition, the binaries are statically linked, which allows the linker to perform further optimizations to reduce the number of run-time constants that are loaded during execution. These optimizations are similar to the optimizations that OM [SrWa93] performs.

The few floating point load instructions contained in the binaries are also taken into account and loads to the zero-registers (R31 and F31) as well as load immediate instructions (LDA and LDAH) are ignored since they do not access the memory and therefore do not need to be predicted.

Table 3.2 summarizes relevant information about the SPECint95 programs. It shows the number of total instructions and load instructions executed as well as the static number of instructions and load instructions contained in the binaries. The numbers in parentheses indicate the percentage of all instructions that are loads. The static counts are in thousands (k) and the dynamic counts in millions (M). The five rightmost columns of the table reflect several kinds of load value predictability (see also Chapter 4).

Information about the SPECint95 Benchmark Suite											
	dynamic			static			predictability (%)				
program	insts	loads	%lds	insts	loads	%lds	reg	lv	st2d	l4v	fcm
compress	60,156 M	10,537 M	(17.5)	22 k	4 k	(17.9)	9.0	40.4	65.8	41.3	35.9
gcc	334 M	80 M	(23.9)	337 k	73 k	(21.6)	19.9	48.5	49.8	65.6	52.0
go	35,971 M	8,764 M	(24.4)	81 k	16 k	(20.1)	9.2	45.9	47.2	64.0	44.7
ijpeg	41,579 M	7,141 M	(17.2)	70 k	14 k	(19.8)	9.4	47.5	47.7	54.1	45.4
li	66,613 M	17,792 M	(26.7)	37 k	7 k	(18.2)	14.3	43.4	50.4	63.8	60.8
m88ksim	82,810 M	14,849 M	(17.9)	51 k	9 k	(17.4)	29.9	76.1	80.0	83.4	79.6
perl	19,934 M	6,207 M	(31.1)	105 k	21 k	(20.3)	19.8	50.7	51.4	80.6	70.8
vortex	95,791 M	22,471 M	(23.5)	161 k	32 k	(20.0)	17.8	65.7	66.0	78.6	66.2
total	403,188 M	87,842 M		864 k	176 k						
average	50,399 M	10,980 M	(21.8)	108 k	22 k	(20.4)	16.2	52.3	57.3	66.4	56.9

Table 3.2: Information about the SPECint95 benchmark suite.

Register predictability “reg” indicates how often the target register of a load instruction already contains the value that the load is about to fetch. Last value predictability “lv” shows how often a load fetches a value that is identical to the previous value fetched by the same load instruction. Stride predictability “st2d” reflects how often a value is loaded that is identical to the last value plus the difference between the last and the second to last value fetched by the same load instruction. Last four value predictability “l4v” indicates how often a value is loaded that is identical to any one of the last four values fetched by the same load. Finally, finite context method predictability “fcm” shows how often a value is loaded that is identical to the value that followed the last time the same last four value sequence was encountered (modulo some hash function). Section 4.3 describes load value predictors that are based on the five presented kinds of predictability. Note that, unlike reg, lv, st2d, and l4v, the fcm predictability results are implementation specific, i.e., they depend on the hash function.

Table 3.2 shows that all eight binaries contain several thousand load instructions (*gcc* contains the most by a large margin). Except for *gcc*, which only compiles the first of its reference input-files, all programs execute several billion load instructions. Despite the high optimization level, the percentage of load instructions is quite high. About every fifth static instruction in the binaries as well as every fifth executed instruction is a load.

The predictability of the load instructions in these programs is also quite high. At least one half of the executed load instructions are theoretically predictable using any method other than “reg”.

3.2.2 Quantile Information

To better estimate how large a load value predictor needs to be, it is important to know how many of the individual load instructions are actually executed and how frequently. Table 3.3 shows the number of load instructions

that contribute the given quantiles (percentages) of all the executed loads in the eight programs. The quantiles are given both in absolute terms as well as in percent of the total number of load sites. For example, the first line in Table 3.3 indicates that of the nearly four thousand load instructions contained in *compress*, only 690 are ever executed (i.e., are executed at least once), which is 17.4% of all the load sites. Furthermore, the 81, 58, and 17 most frequently executed load sites contribute 99, 90, and 50 percent of the dynamically executed loads, respectively.

Quantile Information about the SPECint95 Benchmark Suite									
	load sites	Q100		Q99		Q90		Q50	
compress	3,961	690	(17.4)	81	(2.0)	58	(1.5)	17	(0.4)
gcc	72,941	34,345	(47.1)	14,135	(19.4)	5,380	(7.4)	870	(1.2)
go	16,239	12,334	(76.0)	4,221	(26.0)	1,708	(10.5)	204	(1.3)
ijpeg	13,886	3,456	(24.9)	423	(3.0)	187	(1.3)	42	(0.3)
li	6,694	1,932	(28.9)	312	(4.7)	138	(2.1)	42	(0.6)
m88ksim	8,800	2,677	(30.4)	456	(5.2)	216	(2.5)	52	(0.6)
perl	21,342	3,586	(16.8)	227	(1.1)	169	(0.8)	44	(0.2)
vortex	32,194	16,651	(51.7)	3,305	(10.3)	585	(1.8)	57	(0.2)
average	22,007	9,459	(36.6)	2,895	(9.0)	1,055	(3.5)	166	(0.6)

Table 3.3: SPECint95 quantile information.

The data in Table 3.3 show that a surprisingly small number of load sites contribute most of the executed load instructions. On average, 3.5% of the load sites contribute ninety percent and only 0.6% of the load sites contribute half of all the executed loads. Less than 37% of the load sites are visited at all during execution.

These quantile numbers are promising because they imply that load value predictors do not have to be large enough to store information about all the load sites in a binary. Rather, a predictor capable of only holding nine percent of the load sites can, on average, already handle 99 percent of the dynamically executed loads. Of course, actual predictors need to be designed somewhat larger to handle 99 percent of the executed load instructions due to aliasing and uneven predictor utilization.

3.2.3 Segment Information

For ATOM simulations, the SPECint95 programs are run to completion, resulting in approximately 87.8 billion executed load instructions. However, on the cycle-accurate simulator each benchmark program is only executed for about 300 million instructions (to keep the simulation time reasonable) after having skipped over the initialization code in “fast-simulation” mode. This fast-forwarding is important when only a fraction of a program’s execution can be simulated because the initialization part of a program is not usually representative of the general program behavior [ReCa98]. No instructions are skipped with *gcc* and it is executed for 334 million instructions on the simulator since this amounts to the complete compilation of the first reference input-file. Note that simulating 300 million instructions is an improvement over the 100 million instructions often used for simulations in the current literature [e.g., RFKS98, WaFr97]. Each simulated segment contains over 49 million executed load instructions, which should be sufficient to render any warm-up effects in the load value predictors negligible. Table 3.4 gives information about the simulated segments of each of the eight SPECint95 programs.

Information about the Simulated Segments of the SPECint95 Benchmark Suite												
	skipped	simulated			base	load miss-rate		predictability (%)				
	instrs	instrs	loads	%lds	IPC	L1	L2	reg	lv	st2d	l4v	fcm
compress	5.6 G	300.0 M	53.5 M	(17.8)	1.338	11.72	6.17	13.0	40.7	64.0	41.5	34.6
gcc	0.0 G	334.1 M	79.7 M	(23.9)	1.510	2.39	6.44	19.9	48.5	49.8	65.6	51.9
go	7.0 G	300.0 M	72.1 M	(24.0)	1.414	1.62	15.72	9.4	46.3	48.1	64.5	44.8
ijpeg	2.0 G	300.0 M	49.5 M	(16.5)	1.498	2.31	65.20	9.8	47.5	48.1	55.1	42.8
li	5.0 G	300.0 M	86.4 M	(28.8)	1.911	4.13	0.67	11.7	35.4	41.2	52.4	62.2
m88ksim	2.0 G	300.0 M	62.1 M	(20.7)	1.258	0.13	11.21	49.3	82.3	85.0	88.2	84.3
perl	1.0 G	300.0 M	93.5 M	(31.2)	1.567	0.00	46.87	20.0	50.7	51.4	80.6	70.6
vortex	7.0 G	300.0 M	71.0 M	(23.7)	2.922	2.16	11.99	16.4	65.7	66.3	79.9	69.4
average		304.3 M	71.0 M	(23.3)	1.677	3.06	20.53	18.7	52.1	56.7	66.0	57.6

Table 3.4: Information about the eight simulated program segments.

The table shows the number of instructions (in billions) that are skipped before starting the pipeline-level simulations, the number of simulated instructions and load instructions (in millions), the percentage of the simulated instructions that are loads, the instructions per cycle (IPC) on the baseline

structions that are loads, the instructions per cycle (IPC) on the baseline processor (CPU_{Base}), the L1 data-cache and the L2 cache load miss-rates, and the load value predictability similar to Table 3.2. Note that the number of instructions and loads as well as the predictability shown in Table 3.4 are measured in the CPU's commit stage, meaning that only correct path information is included in the table.

As with the complete executions, the percentage of load instructions executed by the programs is also uniformly high in the simulated segments. About every fifth instruction is a load. With an average IPC of 1.7, this results in one executed load instruction every 2.6 cycles. Given that each executed load accesses the predictor twice, once to request a prediction and once to update the predictor, this amounts to one predictor access every 1.3 cycles on average. When accounting for wrong-path loads and loads that are re-executed, the number turns out to be close to one access per cycle. Since prediction and update requests are not evenly distributed over time, sometimes more than one access per cycle is needed. This is why all the predictors used in this study are banked to support multiple accesses per cycle (Chapter 6).

Note that with the exception of *compress*, the benchmark programs do not have very high L1 data-cache load miss-rates, making it hard for a load value predictor to be effective. Some of the L2 load miss-rates are, on the other hand, quite large. However, since the corresponding number of cache accesses is very small (not shown), the large L2 miss-rates do not have a significant impact on the performance.

The fast-forward points were carefully hand-selected such that the simulated segments would be as representative of the whole program as possible. The segment length is 300 million instructions since this appears to be enough to exhibit "average" program behavior. Longer segments do not yield significantly better results. A comparison of Table 3.2 and Table 3.4 shows that both the percentage of executed instructions that are loads and in par-

ticular the predictability found in the eight segments closely match the respective numbers measured over the entire program executions. Only with *li* and *m88ksim* was the search for a representative segment not very successful. Fortunately, *li*'s segment exhibits too low a predictability and *m88ksim*'s too high a predictability, making the average over the eight programs very close to the average over the complete executions.

3.2.4 Segment Quantile Information

Table 3.5 repeats the quantile study shown in Table 3.3, but only takes instructions from the simulated segments into account.

Quantile Information about the Simulated SPECint95 Segments					
	load sites	Q100	Q99	Q90	Q50
compress	3,961	62 (1.6)	35 (0.9)	28 (0.7)	9 (0.2)
gcc	72,941	34,345 (47.1)	14,135 (19.4)	5,380 (7.4)	870 (1.2)
go	16,239	9,619 (59.2)	3,868 (23.8)	1,719 (10.6)	263 (1.6)
ijpeg	13,886	2,757 (19.9)	379 (2.7)	184 (1.3)	53 (0.4)
li	6,694	419 (6.3)	237 (3.5)	120 (1.8)	43 (0.6)
m88ksim	8,800	747 (8.5)	537 (6.1)	199 (2.3)	25 (0.3)
perl	21,342	1,437 (6.7)	225 (1.1)	167 (0.8)	44 (0.2)
vortex	32,194	1,973 (6.1)	958 (3.0)	355 (1.1)	55 (0.2)
average	22,007	6,420 (19.4)	2,547 (7.6)	1,019 (3.2)	170 (0.6)

Table 3.5: Quantile information about the simulated program segments.

Executing only part of a program usually produces lower quantile numbers, in particular for the high quantiles. This phenomenon is quite apparent in Table 3.5. The Q100 and the Q99 numbers are significantly lower than their counterparts in Table 3.3, whereas the Q90 and the Q50 numbers are quite similar. The good match of the Q90 numbers indicates that the selected segments will likely exercise the load value predictors sufficiently to obtain representative results. The low Q99 and Q100 quantiles mean that the selected segments contain proportionately too few infrequently executed

loads. As a result, below average predictor aliasing has to be expected. Note, however, that techniques exist to keep low-frequency load instructions from influencing the predictor (see Section 9.5.2).

3.3 Speedup

Throughout this dissertation, the term *speedup* denotes how much faster a processor becomes when a load value predictor is added to it.

To obtain the speedup delivered by a load value predictor for a given program, the program is executed on both CPU_{Base} (the baseline processor without a load value predictor) and CPU_{LVP} (the same CPU but with a load value predictor). By definition, the speedup then evaluates to the runtime on CPU_{Base} divided by the runtime on CPU_{LVP}. To be independent of the CPU's clock speed the runtime is usually measured in cycles rather than seconds.

$$speedup = \frac{runtime_{Base}}{runtime_{LVP}} = \frac{cycles_{Base}}{cycles_{LVP}}$$

Since a speedup of one indicates no improvement in performance, the *speedup over baseline* is often easier to understand. It is defined as the regular speedup minus one, making the speedup over baseline positive if the load value predictor improves the execution speed and negative if it slows the execution down. Note that the regular speedup is always positive.

$$speedup\ over\ baseline = speedup - 100\%$$

To better estimate the expected performance improvement that a load value predictor will deliver, the speedup over more than one program is normally measured. This is done because a suite of programs is assumed to exhibit a more "average" program-behavior than an individual program.

Once the individual speedups have been obtained, they need to be com-

bined into a single speedup. Several approaches to combining (averaging) speedups can be found in the literature, the most prominent of which are the *harmonic mean*, the *geometric mean*, and the *arithmetic mean*. The harmonic mean always yields the lowest and therefore the most conservative result. Since the arithmetic mean always produces the highest result, the geometric mean is sometimes used as a compromise.

Intuitively, the combined speedup should be equal to the speedup over the single program *P* that does nothing but run the benchmark programs one after the other (in any order). However, to avoid over-representing longer running programs, *P* must execute all the programs for the same amount of time. This corresponds to weighing (i.e., normalizing) the individual benchmark programs with the inverse of their runtimes.

The runtimes can be normalized for CPU_{Base} or for CPU_{LVP}. If the normalization is done for CPU_{Base}, the combined speedup evaluates to the *harmonic mean* of the individual speedups. If, on the other hand, the normalization is done for CPU_{LVP}, the combined speedup turns out to be the *arithmetic mean* of the individual speedups. The proof can be found in Appendix A.

For example, let us assume a benchmark suite consisting of two programs *A* and *B* that require c_a and c_b cycles, respectively, to execute on CPU_{Base}. Let us further assume a load value predictor *L* that speeds up program *A* by a factor of ten and program *B* by a factor of one (i.e., *B*'s runtime remains the same). The runtimes on CPU_{LVP} are consequently $0.1*c_a$ and c_b.

When normalizing for CPU_{Base}, the combined speedup should be equal to the speedup of the program *P* that executes program *A* $\boldsymbol{c_b}$ times and program *B* $\boldsymbol{c_a}$ times. Doing so takes $c_a*\boldsymbol{c_b}+c_b*\boldsymbol{c_a} = 2*c_a*c_b$ cycles on the baseline CPU (both programs are executed for c_a*c_b cycles). When predictor *L* is added, the total runtime becomes $0.1*c_a*\boldsymbol{c_b}+c_b*\boldsymbol{c_a} = 1.1*c_a*c_b$ cycles. The combined speedup is therefore $2.0/1.1 \approx 1.818$, which is equal to the harmonic mean of the two individual speedups.

When normalizing for CPU_{LVP}, program *P* needs to execute program *A* $\boldsymbol{c_b}$

times and program *B* $\mathbf{0.1*c_a}$ times. This takes $0.1*c_a*\mathbf{c_b}+c_b*\mathbf{0.1*c_a} = 0.2*c_a*c_b$ cycles on CPU_{LVP} (both programs are executed for $0.1*c_a*c_b$ cycles) and $c_a*\mathbf{c_b}+c_b*\mathbf{0.1*c_a} = 1.1*c_a*c_b$ cycles on the baseline processor. The speedup now evaluates to 1.1/0.2 = 5.5, which is the arithmetic mean of the individual speedups. For reference, the geometric mean of the two speedups is about 3.162.

As the example illustrates, normalizing for CPU_{LVP} weighs program *B*, which cannot be sped up by the load value predictor, ten times less heavily than normalizing for CPU_{Base} does (the weights are shown in bold face). In general, the more a program can be sped up the relatively more weight it is given when using the arithmetic mean to compute the combined speedup. Thus, the arithmetic mean speedup assumes the “average” program to contain proportionately more code that benefits from a load value predictor than code that does not. I do not believe this to be a valid assumption, which is why all the averaged speedups presented in this dissertation are harmonic mean speedups.

3.4 Other Metrics

The main metric used in this thesis is the speedup that a load value predictor delivers (previous section). Unfortunately, determining the speedup requires the use of a cycle-accurate simulator, which can be prohibitively slow. Moreover, the speedup is dependent on the architectural features of the underlying CPU and the characteristics of the memory subsystem. Non-implementation specific metrics, on the other hand, are independent of any particular processor architecture and are often quite easy to obtain.

Most load value predictors include some form of *confidence estimator* to help determine how likely a predicted value is to be correct (Chapter 5). If the likelihood for a correct prediction is below a preset threshold, no prediction is attempted. This can significantly reduce the number of mispredictions and

consequently the overhead incurred by the misprediction recovery mechanism.

A load value predictor with a confidence estimator can produce four outcomes for every executed load instruction, as is depicted in Figure 3.1. The number of times each of these four classes is encountered during the execution of a program is referred to as *P+*, *P-*, *N+*, and *N-*.

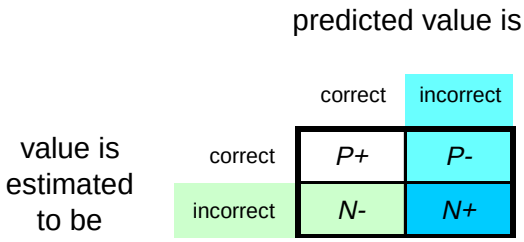

Figure 3.1: The four prediction classifications.

Measuring these four numbers is straightforward. To make the result independent of the total number of executed load instructions, the numbers need to be normalized.

Normalization: (*P+*) + (*P-*) + (*N+*) + (*N-*) = 1

After the normalization, *P+* represents the percentage of all executed load instructions that were correctly predicted. *P-* indicates the percentage of all loads that were mispredicted. *N+* shows what percentage of the dynamically executed load instructions the predictor did not attempt to predict and, if it had, the predicted value would indeed not have been correct. Finally, *N-* is the percentage of all loads the predictor decided not to predict (because of a low confidence) even though the prediction would have been correct.

Unfortunately, the four numbers by themselves do not represent adequate metrics for comparing predictors. For example, it is unclear if predictor *A* is superior to predictor *B* if predictor *A* has both a higher *P+* and a higher *P-*

than predictor *B*, i.e., predictor *A* makes both more correct and more incorrect predictions than predictor *B*.

Fortunately, meaningful metrics for confidence estimation exist that can be derived from these four numbers. These metrics have recently been adapted to and used in the domain of branch prediction and multi-path execution [JRS96, GKMP98]. I adopted the metrics for load value prediction [BuZo99a] with a change in nomenclature. The terms in parentheses represent the standard terminology for diagnostic tests. They are all higher-is-better metrics.

- **Potential**: $POT = (P+) + (N-)$

  The POT represents the percentage of predictable values (predictability).

- **Accuracy** (Predictive Value of a Positive Test): $ACC = \frac{P+}{(P+) + (P-)}$

  The ACC represents the probability that an attempted prediction is correct.

- **Coverage** (Sensitivity): $COV = \frac{P+}{(P+) + (N-)} = \frac{P+}{POT}$

  The COV represents the fraction of predictable values identified as such.

Note that ACC, COV, and POT fully determine *P+*, *P-*, *N+*, and *N-* given that they are normalized.

The potential describes the quality of the value predictor and is independent of the confidence estimator as long as predictor updates are not controlled by the confidence estimator (which is the case throughout this dissertation).

Together, the accuracy and the coverage describe the quality of the confidence estimator. A high accuracy represents a high ratio of correct predictions (which save cycles) over incorrect predictions (which cost cycles) but

usually also means few overall prediction attempts (i.e., the higher the accuracy, the more conservative the predictor). A high coverage, on the other hand, represents a good exploitation of the existing potential, which normally translates into many prediction attempts.

The accuracy and the coverage are antagonistic, meaning that tuning a predictor to increase either one almost always decreases the other. Improving both ACC and COV at the same requires fundamental changes to the predictor's design such as replacing the confidence estimator with a more advanced one (see Chapter 5).

3.5 Summary

The baseline CPU used for the cycle-accurate simulations is configured to closely mimic a DEC Alpha 21264, which is one of the fastest currently available microprocessors. This choice was made to illustrate that even a high-performance CPU can greatly benefit from a load value predictor.

The performance of the various load value predictors discussed in this dissertation is evaluated using the eight integer programs of the SPEC95 benchmark suite. Highly optimized binaries are used to show that load value prediction is beneficial beyond what current optimizing compilers can achieve. A detailed comparison of information about the benchmark suite as a whole and about the program-segments used for the simulations shows that the selected segments are very representative of the complete program executions.

To get as close as possible to measuring the real effectiveness of a load value predictor, speedup results from a cycle-accurate simulator are used almost exclusively as a performance metric in this thesis. Moreover, the speedups over the individual benchmark programs are combined using the harmonic mean so as not to overestimate the performance of the studied load value predictors.

Chapter 4

Context-Based Value Predictors

This chapter introduces the basic architecture common to all context-based load value predictors discussed in this thesis. Furthermore, the architecture of an implementable predictor for each kind of load value predictability mentioned in Section 3.2 (i.e., last value, stride 2-delta, register file, finite context method, and last four value predictability) is presented. The chapter concludes with a performance study of these five predictors.

4.1 Context-Based Value Prediction

Without context, it is almost impossible to predict a load value since a 32-bit word can hold over four billion distinct values and a 64-bit word over 10^{19} values. Even if a highly uneven distribution with only twenty likely load values is assumed, the best we can hope for is still only a five percent prediction accuracy, which is probably too low to be useful.

Fortunately, load values tend to cluster, repeat, occur in iterating sequences, exhibit discernable patterns, and correlate with one another, all of which is referred to as *load value locality*. This locality is the reason why it is not only feasibly but actually quite effective to make predictions based on recently seen load values, i.e., based on context. I use the terms *load value locality* and *load value predictability* interchangeably in this dissertation. Both terms refer to the percentage of load values that can be correctly predicted with a given prediction method.

Based on the numbers shown in Table 3.2, the five measured kinds of

load value locality vary from nine to about 83 percent, depending on the program and the type of locality. These percentages are much higher than the five percent from the example of twenty equally distributed values. Of course, it remains to be determined how much of this potential can actually be exploited by a predictor.

The remainder of this chapter describes the design of five load value predictors that are able to exploit a substantial amount of the existing load value locality.

4.2 Generic Context-Based Load Value Predictor

Figure 4.1 shows the general structure common to all context-based load value predictors that are discussed in this dissertation. A context-based predictor is essentially a direct mapped cache of 2^n lines. Each line retains information about previous executions of one load instruction (modulo the predictor size).

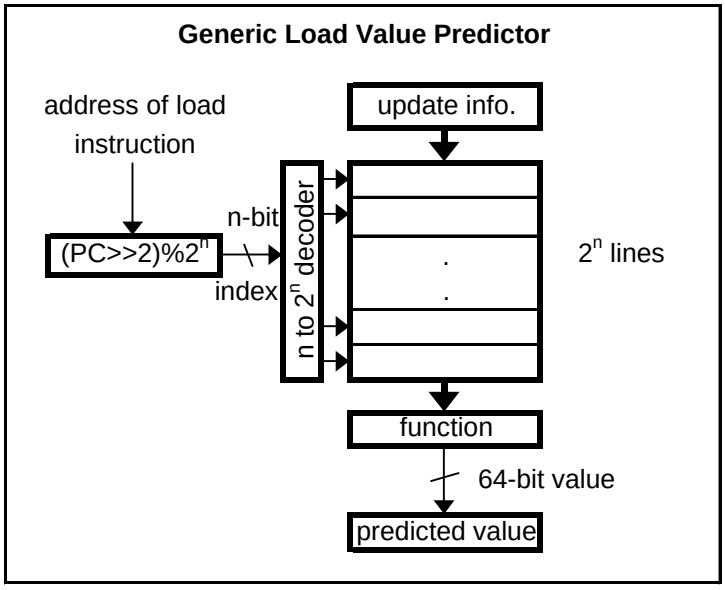

Figure 4.1: The components of a context-based load value predictor.

When a load instruction has to be predicted, the predicted value is computed using the stored information of that load instruction, hence the term context-based. The function that computes the prediction can be as simple as the identity function (last value predictor) and as complex as accessing a lookup table (finite context method predictor). Naturally, accessing the predictor must be faster than accessing the conventional memory since a prediction becomes obsolete as soon as the true load value is available.

Whenever the memory subsystem satisfies a load request, the corresponding predictor line is updated with the true load value and possibly other information. Note that all load instructions, whether they are predictable or not, access the memory and therefore update the predictor.

All the predictors used in this study are direct mapped, that is, the *n* least significant bits of a load instruction's PC (that are not always zero) are used as an index into the predictor to select one of the 2^n predictor lines. Note that the PC of the load instruction is used and not the effective address.

$$\text{index}(PC_{load}) = (PC_{load} >> 2) \% 2^n$$

This is probably the simplest and fastest meaningful hash-function. The ">> 2" eliminates the two least significant bits that are always zero because instructions need to be word aligned in the Alpha processor on which all my measurements are based. Adding a more complex hash-function may result in less aliasing but will most likely increase the length of the critical path. Since direct-mapping results in only little aliasing even with moderate predictor sizes (see Section 5.4), this simple but effective hash-function is frequently used [Gab96, GaMe98, LiSh96, LWS96, SaSm97b, WaFr97]. An investigation of more sophisticated hash-functions (e.g., set associativity) is left for future work.

Note that in cache terminology, direct-mapping implies the presence of tag bits. However, unlike caches load value predictors do not have to be cor-

rect all the time, meaning that tags are not mandatory. Because of the small amount of observed aliasing (see Section 5.4), only partial tags or no tags are often used in predictors to reduce their size. Load instructions that do alias simply have to share a line in the load value predictor, i.e., they overwrite each other's information in the predictor.

4.3 Five Context-Based Load Value Predictors

Based on the generic load value predictor from Figure 4.1, predictors can be built and tailored to exploit different kinds of load value locality by choosing what information to store in them and by performing various kinds of computations with this information. The following subsections show possible implementations of five basic load value predictors to exploit last value, stride 2-delta, register, finite context method, and last four value locality.

4.3.1 Last Value Predictor "LV"

The last value predictor [Gab96, LWS96] (abbreviated as LV) always predicts that a load instruction will load the same value that it did the previous time it was executed. Hence, the only information that needs to be stored in the predictor is the most recently loaded value. Predictions retrieve this value and updates overwrite the stored value with the new load value to make it available for the next prediction.

The last value predictor's operation can formally be described with the following pseudo-code, where the numeral subscripts indicate the size in number of bits, "load" refers to the load instruction being predicted or updated, and "value" is either the predicted value or the update value. The first line, which describes the predictor, lists the fields that make up a predictor line inside the curly brackets as well as the number of predictor lines. In this in-

stance, each predictor line contains a single field called “last_value” that is 64-bits wide and there are 2^n such lines.

predictor: $\{last\_value_{64}\} \bullet 2^n$

prediction: $value = last\_value[index(PC_{load})];$

update: $last\_value[index(PC_{load})] = value;$

predictable sequences:

- repeating values, e.g., 3, 3, 3, 3, ...

Figure 4.2 illustrates the structure of this predictor. The number 64 in the first predictor line denotes the width of this field, i.e., every line contains a 64-bit wide field to store the last value.

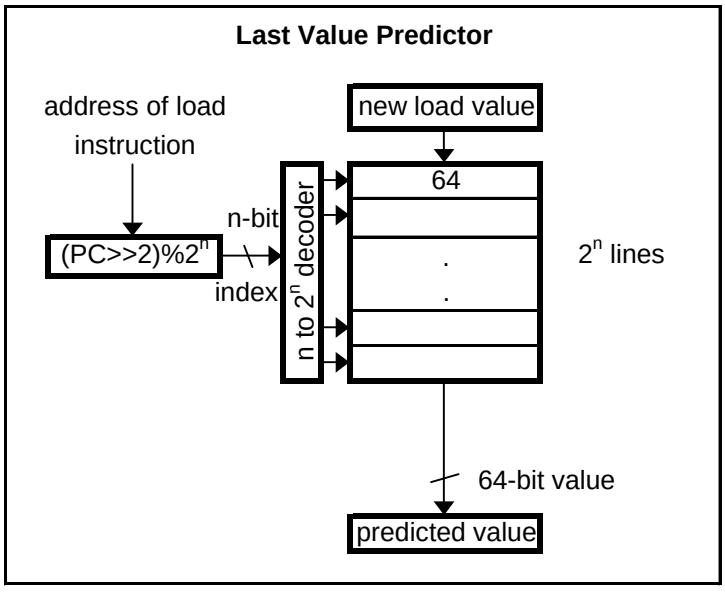

Figure 4.2: The last value predictor.

4.3.2 Register Predictor “Reg”

The register predictor [TuSe99] (abbreviated as Reg) is even easier to implement. Since it always predicts that the target register of the load instruction (the register that is about to receive the loaded value) already contains the correct load value even before the load is executed, no information has to be stored in the predictor. In Chapter 5 we will see that this predictor still needs to store some information to work well. Figure 4.3 illustrates the structure of the register predictor.

Note that none of the benchmark programs used were compiled with this (or any other) kind of load value predictor in mind. Consequently, the predictability for this predictor is not very high. However, the register predictability can be improved upon by modifying the register allocator [TuSe99].

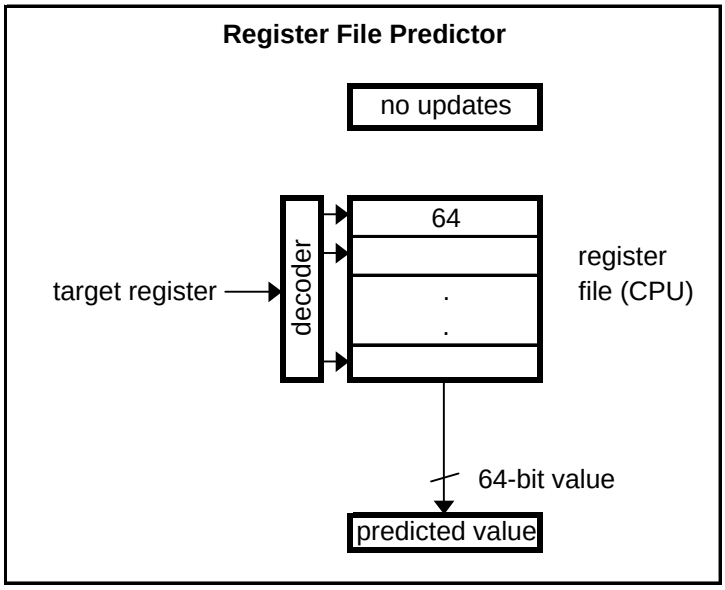

Figure 4.3: The register predictor.

predictor: { } (CPU’s register file is used as source of values)

prediction: value = register[target(load)];

update: no operation

predictable sequences:

- depends on register allocator

4.3.3 Stride 2-delta Predictor “St2d”

The stride predictor [Gab96] (abbreviated as St) truly computes the predicted value and is therefore able to predict never before seen values.

In its conventional form, this predictor stores the last value along with the difference (called the stride) between the last and the second to last loaded value. The stride is added to the last value when a prediction is made to form the predicted value. Once the true load value is available, the predictor’s stride field is updated to reflect the difference between the last value (which is stored in the predictor) and the true load value. Then the last value stored in the predictor is replaced with the new load value. Since about 98% of all the observed strides fall within the range of –128 to 127 [RFKS98], eight bits per predictor line are sufficient to capture almost all strides.

A last value predictor can only predict sequences of constant values. Such sequences can, however, also be predicted with a stride predictor. The stride is simply zero in this case. Hence, the stride predictor might be regarded as a superset of the last value predictor. There exists, however, a subtle difference between the two predictors. A closer look at the following sequence of load values illustrates this difference.

. . . A A A B B B C C C . . .

If we assume that both a stride and a last value predictor have been predicting Xs before reaching the first A of the above sequence (where X≠A, A≠B, and B≠C), we find that the last value predictor predicts six out of the nine values correctly (66.7%), whereas the stride predictor only gets three

values right (33.3%). Evidently, the last value predictor is superior to the stride predictor for sequences of repeating values and particularly for short sequences of repeating values. The reason is that the last value predictor makes one mistake per transition in the sequence and the stride predictor makes two.

This is indeed a problem in practice because programs fetch a surprisingly large number of short sequences of repeating values. For instance, Figure 4.4 shows the percentage of dynamically executed load instructions that fetch sequences of repeating values of the given lengths. The numbers are averages over the eight SPECint95 programs.

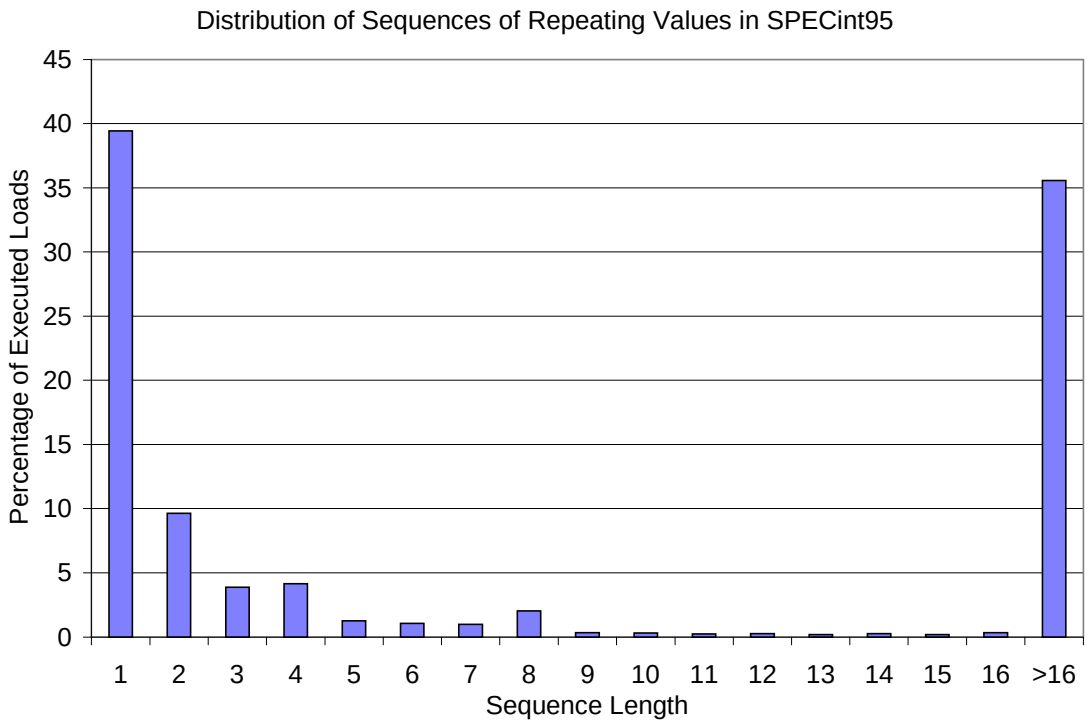

Figure 4.4: Average run-length of sequences of repeating load values.

As the figure illustrates, a considerable percentage of loads fetch sequences of between two and eight repeating values. In fact, the number of short sequences comprising only two, three, or four repeating values is so

large in comparison to sequences that exhibit true stride behavior (i.e., have a non-zero stride) that the stride value locality is smaller than the last value locality for six out of the eight SPECint95 programs, as Table 4.1 shows (the corresponding six stride load value localities are shown in bold print). Only *li* and *compress* exhibit more stride than last value locality.

	compress	gcc	go	ijpeg	li	m88ksim	perl	vortex	average
last value locality (%)	40.4	48.5	45.9	47.5	43.4	76.1	50.7	65.7	52.3
normal stride locality (%)	61.3	**41.3**	**38.3**	**37.6**	44.1	**76.1**	**42.8**	**60.3**	50.2
stride 2-delta locality (%)	65.8	49.8	47.2	47.7	50.4	80.0	51.4	66.0	57.3

Table 4.1: Last value, stride, and stride 2-delta load value locality.

To remedy this shortcoming, the more sophisticated stride 2-delta predictor has been proposed [SaSm97a] (abbreviated as St2d). The 2-delta refers to the fact that this predictor retains two strides instead of only one. The first stride is identical to the one found in the conventional stride predictor. The second stride is only updated if the current update-stride is the same as the stride already stored in the first stride field. In other words, the second stride is only updated if the same stride has been seen at least twice in a row. Only the second stride is used for computing the predicted value.

This scheme effectively eliminates the problem of making two consecutive mispredictions upon a sequence change from one sequence of constant values to another because the working stride (the second stride) remains zero during this transition. As the last line of Table 4.1 shows, the stride 2-delta load value locality is higher than the last value locality for all eight SPECint95 programs. Of course the second stride field also only needs to be eight bits wide, as is outlined in Figure 4.5. In the pseudo code below, the function $LSB_{0..7}(x)$ extracts the eight least significant bits of x. Unless otherwise noted, all stride predictor results in this dissertation refer to the more sophisticated stride 2-delta predictor.

predictor: $\{last\_value_{64}, stride1_8, stride2_8\} \bullet 2^n$

prediction: value = last_value[index(PC_{load})] + stride2[index(PC_{load})];

update: temp = $LSB_{0..7}$(value – last_value[index(PC_{load})]);
if (temp == stride1[index(PC_{load})]) stride2[index(PC_{load})] = temp;
stride1[index(PC_{load})] = temp;
last_value[index(PC_{load})] = value;

predictable sequences:

- repeating values, e.g., -2, -2, -2, -2, ...
- constant strides, e.g., -4, -2, 0, 2, 4, ...

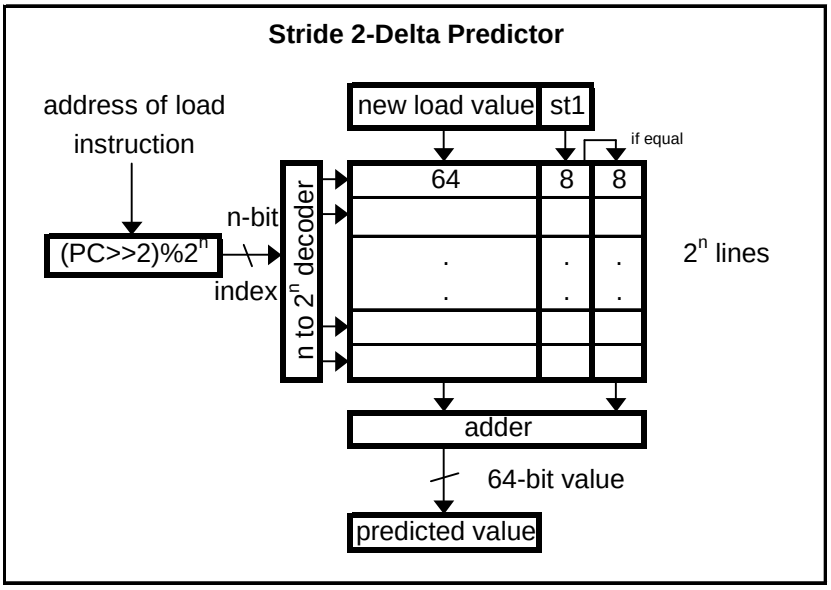

Figure 4.5: The stride 2-delta predictor.

4.3.4 Last Four Value Predictor “L4V”

The last four value predictor [BuZo99b, LiSh96] (abbreviated as L4V) is similar to the last value predictor except every predictor line retains the four most recently loaded values instead of only the most recent value. Chapter 7 discusses last *n* value predictors in more detail and shows that storing the last four values results in good performance.

Deciding which one of the four values to use for a prediction is the job of the selector, which is described in Section 5.4.1. The last four value predictor can be thought of as four independent last value predictors operating in parallel and a meta-predictor that chooses which predictor to believe. Figure 4.6 illustrates this. The last four value predictor therefore represents a hybrid predictor. Hybrid predictors are the topic of Chapter 8.

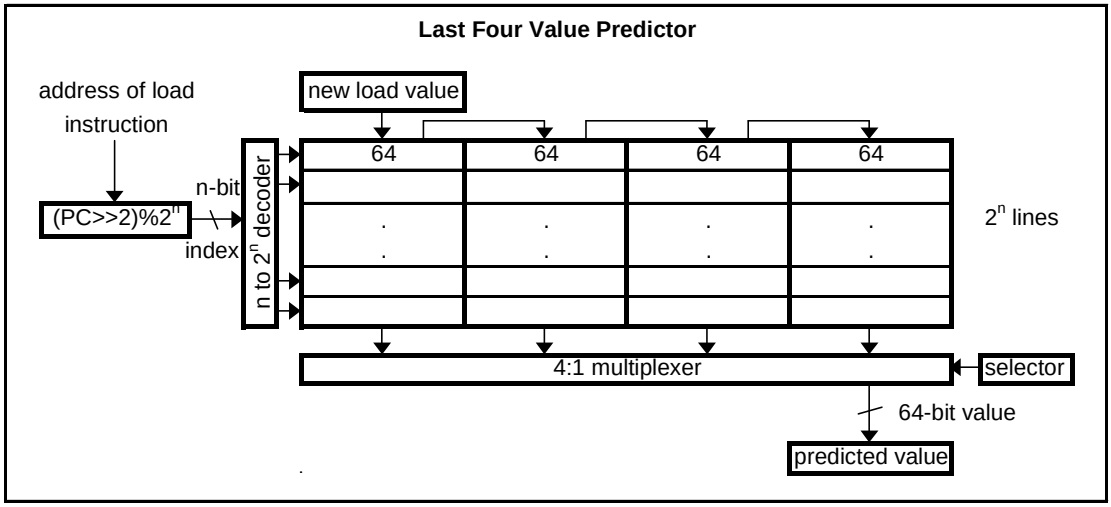

Figure 4.6: The last four value predictor.

The difference between the last four value and a conventional hybrid predictor, and the reason why the last four value predictor is included here, is that a normal hybrid consists of different predictors that are updated with the same information whereas the last four value predictor comprises four identi-

cal predictors that are updated with different information. Section 5.4.1 examines the last four value predictor in more detail and explains the select function used below.

predictor: $\{last\_value1_{64}, last\_value2_{64}, last\_value3_{64}, last\_value4_{64}\} \bullet 2^n$

prediction: value = select(last_value1[index(PC_{load})],
last_value2[index(PC_{load})], last_value3[index(PC_{load})],
last_value4[index(PC_{load})]);

update: last_value4[index(PC_{load})] = last_value3[index(PC_{load})];
last_value3[index(PC_{load})] = last_value2[index(PC_{load})];
last_value2[index(PC_{load})] = last_value1[index(PC_{load})];
last_value1[index(PC_{load})] = value;

predictable sequences:
- repeating values, e.g., 2, 2, 2, 2, ...
- alternating values, e.g., -1, 0, -1, 0, -1, ...
- short repeating cycles, e.g., 1, 2, 3, 1, 2, 3, 1, ...

4.3.5 Finite Context Method Predictor “FCM”

The most complex and sophisticated non-hybrid predictor is the finite context method predictor [SaSm97a, SaSm97b] (abbreviated as FCM). It is similar to the last four value predictor in so far as that it retains the last four loaded values in every predictor line. However, since these values are only used to compute an index into the predictor’s second level (a lookup table), they are not stored in their full length but rather in a more compact, preprocessed form. The second level, a 2048-entry direct mapped, tag-less cache, stores the values that follow every seen sequences of four last values (modulo the table size). Since the second level is shared, load instructions

can communicate information to other loads in this predictor. Hence, after fetching a sequence of arbitrary load values, which warms up the finite context method predictor, the same sequence can be predicted correctly even if it is fetched again by a different load instruction.

For this study, the size of the second level of the FCM predictor is fixed at 2048 entries and the index into the second level is computed as follows.

$$\text{hash}(val) = val_{63..56} \oplus val_{55..48} \oplus val_{47..40} \oplus val_{39..32} \oplus val_{31..24} \oplus val_{23..16} \oplus val_{15..8} \oplus val_{7..0};$$

$$\text{index2}(val1, val2, val3, val4) = \text{hash}(val1) \oplus \text{hash}(val2)*2 \oplus \text{hash}(val3)*4 \oplus \text{hash}(val4)*8;$$

$$\text{line} = \text{index2}(last\_value1, last\_value2, last\_value3, last\_value4);$$

The "⊕" in the above formulas represents the logical exclusive-or function. The presented index2 function is similar to the functions used by other people for the finite context method predictor [ReCa98, RFKS98, SaSm97b]. It utilizes all 64 bits of the load values for computing the index. Furthermore, the values are shifted relative to one another so that sequences of constant values do not cancel each other out (i.e., always yield an index of zero) when they are exclusive-or'ed.

Another benefit of the above function is that part of it (the first line) can be evaluated before the information is inserted into the first level of the predictor. Doing so significantly reduces the size of the predictor. Since hash(val) always yields an eight-bit result, each line in the first level of the predictor only needs to store four eight-bit values instead of the four 64-bit values, as is illustrated in Figure 4.7.

level1: $\{hash1_8, hash2_8, hash3_8, hash4_8\} \bullet 2^n$

level2: $\{FCMvalue_{64}\} \bullet 2048$

predictor: level1 + level2

prediction: $line = hash1[index(PC_{load})] \oplus hash2[index(PC_{load})]*2 \oplus hash3[index(PC_{load})]*4 \oplus hash4[index(PC_{load})]*8;$

$value = FCMvalue[line];$

update: $line = hash1[index(PC_{load})] \oplus hash2[index(PC_{load})]*2 \oplus hash3[index(PC_{load})]*4 \oplus hash4[index(PC_{load})]*8;$

$FCMvalue[line] = value;$

$hash4[index(PC_{load})] = hash3[index(PC_{load})];$

$hash3[index(PC_{load})] = hash2[index(PC_{load})];$

$hash2[index(PC_{load})] = hash1[index(PC_{load})];$

$hash1[index(PC_{load})] = hash(value);$

predictable sequences:

- reoccurring values, e.g., 3, 7, 4, 9, 2, ..., 3, 7, 4, 9, 2, ...
- addresses loaded during the traversal of dynamic data structures

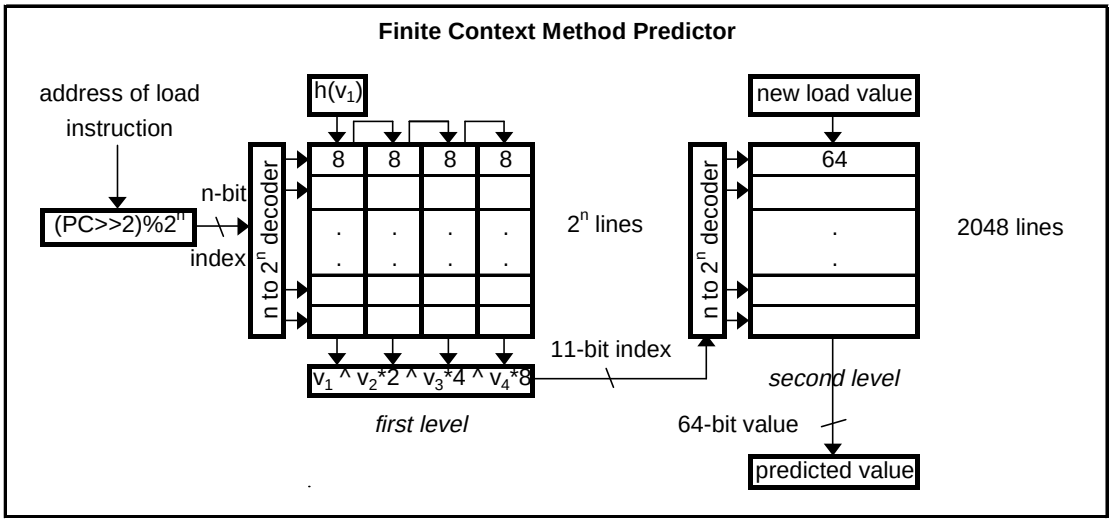

Figure 4.7: The finite context method predictor.

4.4 Predictor Performance

Figure 4.8 shows the harmonic-mean speedup over SPECint95 delivered by the five predictors from the previous sections. For these measurements, the predictors are 2048 lines tall and include an eight-bit partial tag in each line. In the FCM predictor, both levels comprise 2048 lines. No prediction is attempted in case of a tag miss. Furthermore, each predictor is divided into four independent banks (see Chapter 6).

Two speedup numbers are given for each predictor, one showing the speedup over the baseline processor when a re-fetch misprediction recovery mechanism is used and the other when a re-execute recovery mechanism is used (see Chapter 5). Since the presented predictors vary greatly in size, the presented results should not be used for inter-predictor comparisons.

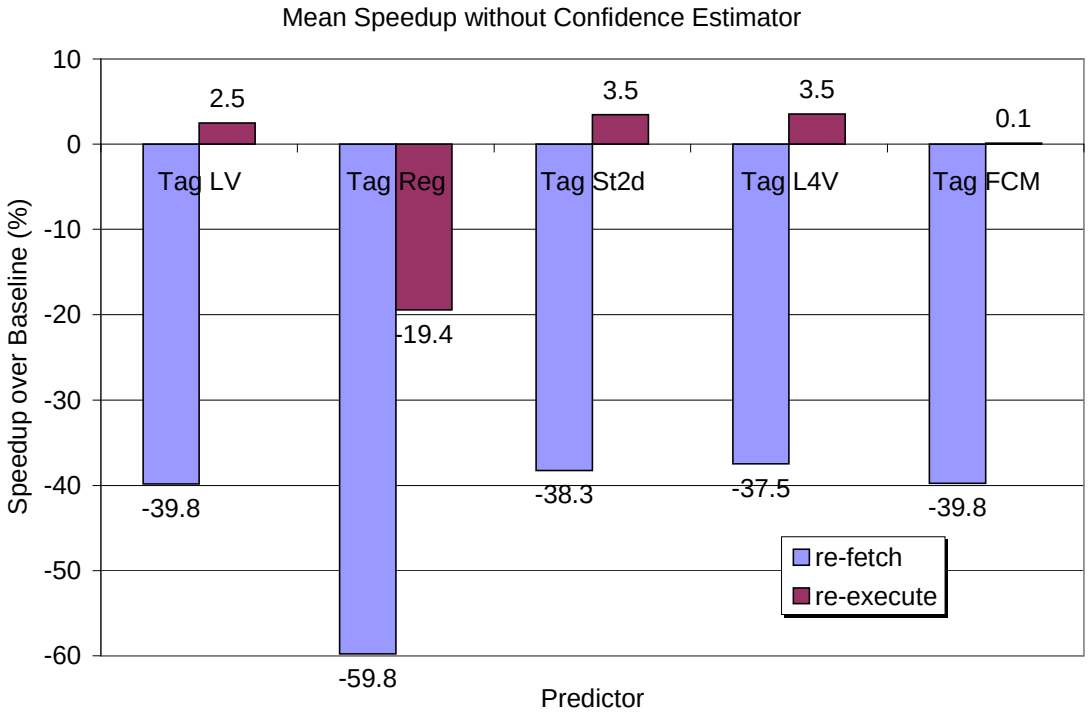

Figure 4.8: Mean speedup of five context-based predictors.

Clearly, the performance of the five predictors is rather poor. In fact, all the predictors slow the processor down with a re-fetch misprediction recovery mechanism. With re-execute, which incurs smaller penalties than re-fetch, the register value predictor slows down the SPECint95 programs on average while the remaining four predictors yield a small positive speedup.

The reason for the poor performance is the large number of mispredictions. The average load value locality shown in Table 3.2 is roughly fifty percent, meaning that a large fraction of loads cannot be correctly predicted with these prediction techniques. Trying to predict these loads will inevitably result in mispredictions.

Because incorrect predictions necessitate a recovery process and thus incur a cycle penalty, a high misprediction-rate may result in more added cycles than are saved by the correct predictions. With re-fetch, the misprediction penalties do indeed more than offset the benefit of the correct predictions, which is why the overall performance decreases.

The next chapter introduces confidence estimators, which are able to reduce the number of mispredictions and thus improve the performance of all five predictors. In fact, with confidence estimators, all five predictors yield positive speedups even with re-fetch recovery.

4.5 Summary

This chapter introduces the concept of context-based load value prediction as well as the architecture of such predictors. The functionality is explained and a possible implementation is given for a last value, a register, a stride 2-delta, a last four value, and a finite context method predictor.

A performance evaluation of these five predictors shows that all of them make too many costly mispredictions to be effective. In the next chapter a technique is discussed that can eliminate most mispredictions, which boosts the performance of these five predictors, making them profitable to employ.

Chapter 5

Confidence Estimators

This chapter introduces and evaluates two commonly used dynamic confidence estimators. First, the *bimodal* confidence estimator is described, which is based on saturating counters. Then the prediction outcome history-based *SAg* confidence estimator I developed is derived. It is more complex but performs better than the *bimodal* confidence estimator on certain kinds of load value sequences, as sample traces illustrate. The performance of both confidence estimators in connection with the five load value predictors from the previous chapter is evaluated. Finally, the expressiveness of non-speedup metrics is studied.

5.1 The Need for Confidence Estimators

As the speedup results from Section 4.4 illustrate, context-based load value predictors are quite ineffective in spite of the relatively high load value predictability (see Table 3.2). As mentioned, the problem is the high cost of recovering from the many mispredicted loads, which can more than eradicate the performance advantage derived from the correctly predicted loads.

Fortunately, making no prediction does not incur a cycle penalty, meaning that it is better not to make a prediction than to make an incorrect prediction. Consequently, identifying predictions that are likely to be incorrect and inhibiting them can reduce the number of penalty cycles and thus improve the predictor performance.

This is why almost all load value predictors include some form of *confi-*

dence estimator (CE) to estimate how likely a prediction is to be correct [CRT99, LWS96, ReCa98, RFKS98, SaSm97b, TuSe99, WaFr97]. Predictions are only allowed if the estimated confidence is high (i.e., above a preset threshold). The two main approaches for dynamic confidence estimation in the domain of value prediction, *saturating counters* and *prediction outcome histories*, are described in the subsequent sections. The latter was developed by me to overcome a deficiency of the former confidence estimator, as discussed in Section 5.2.1.

5.2 The *Bimodal* Confidence Estimator

One way of estimating the likelihood of a correct load value prediction is to count how often the load was correctly predicted in the (recent) past. The intuition behind this approach is that the past behavior tends to be indicative of what will happen in the near future. For example, if a load was predicted successfully most of the time in the recent past, there is a good chance that its next prediction will be successful, too. Therefore, counting the number of times a predictor was able to predict a load instruction correctly yields a measure of confidence where a higher count implies a higher probability that the next prediction will be correct.

Note that it is vital to count both the correct prediction and the times the predictor could have made a correct prediction but was not allowed to do so due to a low confidence at the time. If only the attempted predictions that turned out to be correct are counted, then the confidence cannot recover once it has reached a point that inhibits further predictions.

A commonly used hardware device for counting events is the *saturating up/down counter*. A saturating counter can count up and down within two boundaries, say zero and fifteen. Once the counter has reached fifteen, counting up will not change its value. Likewise, counting down from zero leaves the counter at zero. Such a counter is said to have a bottom of zero

and a top of sixteen. Note that while the bottom value is the lowest reachable value, the top value cannot actually be reached. The highest reachable value is top-1. This definition of top and bottom simplifies many aspects of dealing with saturating counters. Note also that the counter *increment* and *decrement* (the latter is often referred to as *penalty*) can be an integer greater than one. In particular, penalties above one are frequently useful in connection with load value predictors.

The *bimodal* confidence estimator is based on saturating up/down counters that record how many predictable values were encountered in the recent past. I adopted the name "*bimodal*" from the structurally identical bimodal branch predictor [McF93].

The higher the counter value in a *bimodal* CE, the higher the confidence that the next prediction will also be correct since predictable load instructions are assumed not to suddenly become unpredictable and vice-versa. A preset threshold value determines when the confidence estimator allows predictions to take place and when it does not. For instance, a threshold of ten allows predictions as long as the value of the saturating counter does not fall below ten. If it does, further predictions are inhibited until the counter value rises to ten or higher again.

A saturating counter-based confidence estimator can be described with five parameters: the top value, the bottom value, the increment, the decrement or penalty, and the prediction threshold. Only powers of two are used as the top value in this dissertation. This is not mandatory but it limits the large search space somewhat. By definition, the counter value is always lower than the top value. The bottom value, the lower bound for the counter value, is always zero throughout this thesis. This is not a restriction because the top, threshold, and bottom values can always be shifted to make the bottom value zero without loss of generality. The increment is always one since not even a single preliminary experiment of mine has shown larger increments to be useful. The current literature also only uses increments of one

[ReCa98, RFKS98, WaFr97]. Finally, the threshold and penalty values are determined individually for each predictor and misprediction recovery mechanism, as is the top value. Since the smallest number of bits required to store any value in the range [0..top) is $\lceil \log_2(top) \rceil$, the top value determines the width of the counters. Frequently used top values in this dissertation are eight and sixteen, making the counters three to four bits wide.

Figure 5.1 illustrates the structure of the *bimodal* confidence estimator when added to a generic load value predictor. The shaded components make up the confidence estimator. "c" indicates a saturating up/down counter.

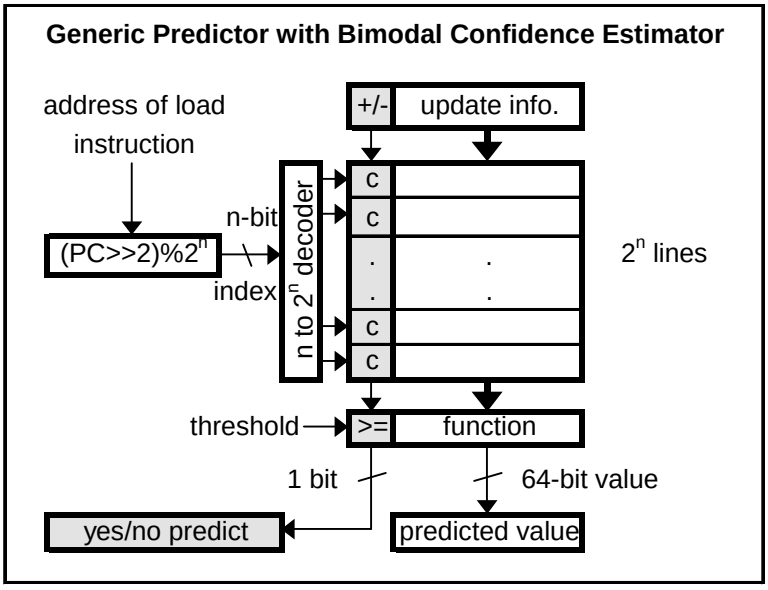

Figure 5.1: The *bimodal* confidence estimator (shaded).

Note that each line of the predictor contains its own saturating counter, meaning that the confidence of each line is measured separately. As long as only one load instruction is mapped to each line in the predictor, each load's confidence is measured individually. If multiple loads alias in the same predictor line, the resulting counter value is a combination of the confidences of the loads that share the line.

Except for the saturating counters, the confidence estimator only consists of a comparator that checks whether the selected counter value is at or above the preset threshold. This can be done in parallel with the prediction of the load value. The predicted value is only used if the confidence is high enough.

When the value predictor is updated, it first makes another prediction whose result is compared with the true load value. If the two values are identical, the saturating counter in the selected predictor line is incremented by one, otherwise it is decremented by the preset penalty.

A *bimodal* confidence estimator's operation can be described with the following pseudo-code. "x" is usually three or four (actually, $x = \lceil \log_2(top) \rceil$).

conf_estim: $\{counter\_value_x\} \bullet 2^n$

prediction: predict = (counter_value[index(PC_{load})] >= threshold);

update: counter_value[index(PC_{load})] = (predicted_value == true_value) ?
min(top-1, counter_value[index(PC_{load})]+1) :
max(0, counter_value[index(PC_{load})]-1);

predictable sequences:

- sequences that do not frequently change from being predictable to being unpredictable and vice-versa

5.2.1 Behavior Study

Predictors with a *bimodal* CE yield good performance (see Section 5.4). Nevertheless, due to the counter-hysteresis, the *bimodal* CE performs poorly on certain kinds of load value sequences. In particular, it cannot adapt to quickly alternating patterns of predictable and unpredictable loads.

As was already shown in the discussion about the stride and the stride

2-delta predictor (Section 4.3.3), programs fetch a surprisingly large number of short sequences of repeating load values (Figure 4.4). Such sequences pose a problem for a *bimodal* CE. To illustrate this, let us assume a last value predictor with a *bimodal* confidence estimator that predicts the following infinite sequence of three repeating values (where A≠B, B≠C, etc.).

. . . A A A B B B C C C . . .

For a two-bit saturating counter (i.e., a counter with a top of four), we find that depending on the penalty and the counter value at the beginning of the sequence, one of three possible patterns of indefinitely repeating counter-values emerges quickly. The three patterns are “2, 3, 3”, “0, 1, 2”, and “1, 2, 3” and are shown in the middle of Table 5.1 in bold print. The left side of the table indicates which initial counter values and penalties result in which pattern. The right side of the table shows the number of prediction attempts, inhibited predictions, correct predictions, and incorrect predictions the *bimodal* CE makes (per three load instructions) depending on the threshold value.

initial value	pen-alty	last value predictable: yes **B**	no **C**	yes **C**	yes **C**	no **D**	yes **D**	case	thresh-hold value	predic-tions	no predic-tions	correct predic-tions	wrong predic-tions
0, 1, 2, 3	1	…	3	**2**	**3**	**3**	2	…	C1	1	3	0	2	1
									C2	2	3	0	2	1
									C3	3	2	1	1	1
0, 1, 2	2	…	2	**0**	**1**	**2**	0	…	C4	1	2	1	1	1
									C5	2	1	2	0	1
0, 1, 2, 3	3								C6	3	0	3	0	0
3	2	…	3	**1**	**2**	**3**	1	…	C7	1	3	0	2	1
									C8	2	2	1	1	1
									C9	3	1	2	0	1

Table 5.1: Behavior study of a *bimodal* confidence estimator.

Note that a last value predictor without a CE would make three prediction attempts and get two of the three load values right and one wrong. Hence, the *bimodal* CE does not help in the cases C1, C2, and C7. In the cases C3, C4, and C8 the CE inhibits one of the correct predictions and none of the in-

correct predictions, making the resulting performance worse than it would be without a CE. The cases C5 and C9 are even worse since the CE inhibits all the correct predictions and only allows the incorrect prediction to take place. Finally, case C6 allows no predictions at all, meaning that the load value predictor is not used.

Clearly, in all nine cases the *bimodal* CE either does not help or makes things worse. Since short sequences of repeating load values make up a considerable part of the observed load value sequences (Figure 4.4), this unfortunate behavior of the *bimodal* confidence estimator is a problem.

Note that a one-bit counter would result in the same prediction behavior as case C4 and three-bit and wider counters always end up in one of the four described outcomes, as the following thought experiment illustrates.

The two cases where the confidence estimator allows no predictions or forces all values to be predicted are uninteresting. In all the remaining two cases (with one or two predictions per three values), some of the values are predicted and some are not. If the third of the three repeating values is predicted (i.e., the counter has reached the threshold), then any *bimodal* predictor will also predict the following value, which happens to be the unpredictable one, because the third value is predictable and will therefore increment the counter, meaning that the next time the counter is queried it will again be above the threshold and cause a prediction. Likewise, if the first of the three repeating values is not predicted (i.e., the counter value is below the threshold), then any *bimodal* predictor will also not attempt to predict the next value, which in our example is predictable, because the first value is unpredictable and will consequently decrement the counter, meaning that the next time the counter is queried it will still be below the threshold and therefore inhibit a prediction. In other words, it is impossible for a *bimodal* confidence estimator to predict the last predictable value and not predict the following unpredictable value and it is also impossible not to predict an unpredictable value while at the same time predicting the following predictable value. As a conse-

quence, only the following four scenarios for the three repeating values are possible: none of them are predicted, only one value is predicted, which has to be the unpredictable one, two values are predicted, of which one has to be the unpredictable value, or all three values are predicted. Consequently, any *bimodal* CE is, independent of its threshold and top value, incapable of predicting the predictable values without also predicting the unpredictable value in the above sample sequence. Hence, the observed deficiency is intrinsic to the *bimodal* CE and cannot be overcome by adjusting parameters.

5.3 The *SAg* Confidence Estimator

Based on the observations made in the previous section, designing a confidence estimator that does not suffer from the described deficiency may be worthwhile. Ideally, a CE should predict the two predictable values and inhibit every third prediction in the example of three repeating load values from the previous section. To recognize the iterating pattern of two predictions followed by one non-prediction, some sort of a history mechanism is probably necessary.

Since confidence estimators are similar to branch predictors, I turned to the branch prediction literature to find an alternative approach that is history-based. One successful idea in branch prediction is keeping a small history recording in which direction each branch recently went [LeSm84]. This idea was later refined to retaining the most recent prediction outcome (success or failure) [SCAP97] rather than the branch direction, which makes the approach useful as a confidence estimator.

In such a CE, the outcome of a prediction is stored in a bit-pattern (called a *history*) where the n^{th} bit represents the outcome of the n^{th} last prediction. Usually a one is used to encode a successful prediction and a zero to encode a misprediction.

Whenever the memory returns a load value, the true load value is com-

pared with its predicted value (even if the prediction was not used) and the outcome of this comparison is shifted into the history, whereby all the bits in the history are shifted by one position and the oldest bit is lost.

If such histories are to be used as a measure of confidence, it is essential to know which ones are (frequently) followed by a correct prediction and which ones are not. The branch prediction literature describes algorithms to accomplish this. For instance, Sechrest et al. [SLM95] suggest profiling a set of programs to record the behavior.

Table 5.2 shows the average SPECint95 last value predictability following each of the sixteen possible four-bit prediction outcome history. For example, the second row of the table states that a *failure, failure, failure, success* history (denoted as *0001*) is followed by a successful last value prediction 26.9% of the time. In this history, *success* denotes the outcome of the most recent prediction. Of all the encountered histories, 2.7% were *0001*.

SPECint95 Last Value Predictability		
history	predictability (%)	occurrence (%)
0000	6.9	32.2
0001	26.9	2.7
0010	19.1	2.9
0011	49.9	1.6
0100	34.3	2.9
0101	33.6	1.9
0110	44.9	1.3
0111	59.4	2.2
1000	24.2	2.7
1001	46.3	1.8
1010	66.8	1.9
1011	66.1	1.9
1100	53.1	1.6
1101	57.2	1.9
1110	52.3	2.2
1111	96.6	38.3

Table 5.2: History-pattern frequency and last value predictability.

Note that it is not necessary to make a prediction following every history with a greater than fifty percent probability of resulting in a correct prediction.

Rather, the predictable/not-predictable threshold can be set anywhere. The optimal setting strongly depends on the characteristics of the CPU the prediction is going to be made on [BuZo99a].

If only a small cost is associated with making a misprediction (as is the case with a re-execute recovery mechanism), it is most likely wiser to predict a larger number of load values, albeit also a somewhat larger number of incorrect ones. If, on the other hand, the misprediction penalty is high and should therefore be avoided (as is the case with re-fetch recovery), it makes more sense not to predict quite as many loads but to be very confident that the ones that are predicted will be correct.

If we want to be highly confident that a prediction is correct, say at least ninety percent confident, the history-based CE would only allow predictions for histories whose predictability is greater than ninety percent, i.e., only for history *1111* based on the data in Table 5.2. Such a four-bit history-based confidence estimator can yield a 96.6% prediction accuracy and predicts 38.3% of all loads of the SPECint95 benchmark suite. Longer histories result in even higher accuracies and better coverage [BuZo98a].

Initially, I built a confidence estimator in which the history patterns that should be followed by a prediction have to be preprogrammed using tables similar to Table 5.2 [BuZo98a]. While this confidence estimator (called an *SSg* CE after the structurally identical *SSg* branch predictor [YePa93]) already outperforms its *bimodal* counterpart [BuZo99a], profile-runs are unfortunately necessary to program the history patterns. Furthermore, the *SSg* CE is completely static and cannot *adapt* to changing program behavior.

To remedy these shortcomings, the CE's design needed to be changed so that the table can be maintained in hardware and updated on-the-fly. In other words, the CE has to record how many correct predictions recently followed each of the possible history patterns. Saturating counters are, of course, appropriate for this task.

By letting saturating counters record the number of correct predictions that

followed each history pattern in the recent past, the counter values dynamically assign a confidence to each history and thus continuously adjust which patterns should be followed by a prediction and which ones should not. Predictions are only allowed if the counter value associated with the current prediction outcome history is above a preset threshold.

The architecture of the resulting *SAg* confidence estimator, which is named after the structurally identical *SAg* branch predictor [YePa93] (the naming conventions are explained in Appendix D), is shown in Figure 5.2.

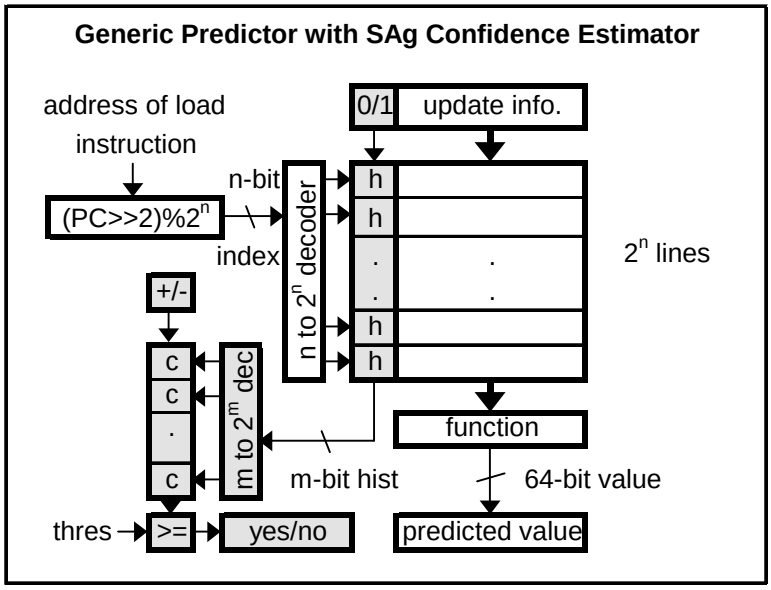

Figure 5.2: The *SAg* confidence estimator (shaded).

The *SAg* CE was developed by me [BuZo98b], and independently by Calder et al. [CRT99]. The following pseudo-code describes its operation. *m* denotes the number of history bits in each predictor line and *x* represents the number of bits in each saturating counter. The prediction outcome histories are marked with 'h' and the saturating counters with 'c'.

CE_level1: $\{history_m\} \bullet 2^n$

CE_level2: $\{counter\_value_x\} \bullet 2^m$

conf_estim: CE_level1 + CE_level2

prediction: predict = (counter_value[history[index(PC_{load})]] >= threshold);

update: counter_value[history[index(PC_{load})]] = (predicted_value == true_value) ?
min(top-1, counter_value[history[index(PC_{load})]]+1) :
max(0, counter_value[history[index(PC_{load})]]-1);
history[index(PC_{load})] = $LSB_{0..m-1}$(history[index(PC_{load})] << 1) | (predicted_value == true_value);

predictable sequences:
- sequences that exhibit (short) repeating behavior

5.4 Performance Comparison

The *SAg* CE requires more hardware than the *bimodal* CE and is most likely slower because of its two-level design. This raises the question whether the extra complexity is worthwhile.

The *SAg* CE has been developed to avoid the poor behavior of the *bimodal* CE on short sequences that alternate between being predictable and being unpredictable. Revisiting the sample sequence from Section 5.2.1 shows that a two-bit history is already sufficient to obtain perfect confidence estimation because the saturating counters associated with the histories 01 and 10 are constantly incremented, meaning that a prediction will be allowed following these histories, the saturating counter associated with history 11 is incessantly decremented and thus inhibits predictions after two consecutive correct predictions, and history 00 never occurs. Hence, the *SAg* CE is able to learn the predictability pattern and can identify the two predictable values and the unpredictable value as such, thus reaping the maximum benefit by predicting all the predictable values and allowing no mispredictions that could

incur cycle-penalties.

To better visualize the behavior and the performance of the two kinds of CEs with a last value predictor, Table 5.3 shows excerpts of real traces from three load instructions found in *gcc* (one of the SPECint95 programs) along with the behavior of four *bimodal* CEs and one *SAg* CE. The first trace (on the left) exhibits almost no last value predictability, the middle trace exhibits almost full last value locality, and the last trace has medium predictability.

Low Last Value Predictability												
	L	CE1		CE1		CE3		CE4		SAg CE		
load value	V	C	P	C	P	C	P	C	P	history	C	P
1074930240		0		0		0		0		________	0	
0		0		0		0		0		________	0	
1074930256		0		0		0		0		________	0	
1074930248		0		0		0		0		________	0	
1074930376		0		0		0		0		________	0	
1074930368		0		0		0		0		________	0	
1074930240		0		0		0		0		________	0	
22		0		0		0		0		________	0	
0		0		0		0		0		________	0	
0	*	0		0		0		0		________	0	
0	*	1		1		1		1		_______o	2	
2		*2*	-	2		*2*	-	2		______oo	0	
0		1		1		0		0		_____oo_	0	
2		0		0		0		0		____oo__	0	
1074908984		0		0		0		0		___oo___	0	
1074908976		0		0		0		0		__oo____	0	
1074909000		0		0		0		0		_oo_____	0	
0		0		0		0		0		oo______	0	
1074908984		0		0		0		0		o_______	0	
1074908976		0		0		0		0		________	1	
1074909000		0		0		0		0		________	0	
10		0		0		0		0		________	0	
1074911312		0		0		0		0		________	0	
0		0		0		0		0		________	0	
1074911312		0		0		0		0		________	0	
6		0		0		0		0		________	0	
1074911568		0		0		0		0		________	0	
0		0		0		0		0		________	0	
1074911568		0		0		0		0		________	0	
6		0		0		0		0		________	0	
correct		**0**		**0**		**0**		**0**			**0**	
missed		2		2		2		2			2	
incorrect		*1*		*0*		*1*		*0*			*0*	

High Last Value Predictability												
	L	CE1		CE1		CE3		CE4		SAg CE		
load value	V	C	P	C	P	C	P	C	P	history	C	P
0	*	**3**	**+**	**3**	**+**	**3**	**+**	**7**	**+**	oooooooo	**15**	**+**
0	*	**3**	**+**	**3**	**+**	**3**	**+**	**7**	**+**	oooooooo	**15**	**+**
0	*	**3**	**+**	**3**	**+**	**3**	**+**	**7**	**+**	oooooooo	**15**	**+**
0	*	**3**	**+**	**3**	**+**	**3**	**+**	**7**	**+**	oooooooo	**15**	**+**
0	*	**3**	**+**	**3**	**+**	**3**	**+**	**7**	**+**	oooooooo	**15**	**+**
0	*	**3**	**+**	**3**	**+**	**3**	**+**	**7**	**+**	oooooooo	**15**	**+**
0	*	**3**	**+**	**3**	**+**	**3**	**+**	**7**	**+**	oooooooo	**15**	**+**
0	*	**3**	**+**	**3**	**+**	**3**	**+**	**7**	**+**	oooooooo	**15**	**+**
0	*	**3**	**+**	**3**	**+**	**3**	**+**	**7**	**+**	oooooooo	**15**	**+**
0	*	**3**	**+**	**3**	**+**	**3**	**+**	**7**	**+**	oooooooo	**15**	**+**
0	*	**3**	**+**	**3**	**+**	**3**	**+**	**7**	**+**	oooooooo	**15**	**+**
0	*	**3**	**+**	**3**	**+**	**3**	**+**	**7**	**+**	oooooooo	**15**	**+**
0	*	**3**	**+**	**3**	**+**	**3**	**+**	**7**	**+**	oooooooo	**15**	**+**
0	*	**3**	**+**	**3**	**+**	**3**	**+**	**7**	**+**	oooooooo	**15**	**+**
0	*	**3**	**+**	**3**	**+**	**3**	**+**	**7**	**+**	oooooooo	**15**	**+**
0	*	**3**	**+**	**3**	**+**	**3**	**+**	**7**	**+**	oooooooo	**15**	**+**
0	*	**3**	**+**	**3**	**+**	**3**	**+**	**7**	**+**	oooooooo	**15**	**+**
0	*	**3**	**+**	**3**	**+**	**3**	**+**	**7**	**+**	oooooooo	**15**	**+**
144		*3*	-	*3*	-	*3*	-	*7*	-	oooooooo	*15*	-
144	*	**2**	**+**	2		1		**5**	**+**	ooooooo_	**15**	**+**
144	*	**3**	**+**	**3**	**+**	**2**	**+**	**6**	**+**	oooooo_o	**15**	**+**
144	*	**3**	**+**	**3**	**+**	**3**	**+**	**7**	**+**	ooooo_oo	**15**	**+**
144	*	**3**	**+**	**3**	**+**	**3**	**+**	**7**	**+**	oooo_ooo	**15**	**+**
144	*	**3**	**+**	**3**	**+**	**3**	**+**	**7**	**+**	ooo_oooo	**15**	**+**
144	*	**3**	**+**	**3**	**+**	**3**	**+**	**7**	**+**	oo_ooooo	**15**	**+**
144	*	**3**	**+**	**3**	**+**	**3**	**+**	**7**	**+**	o_oooooo	**15**	**+**
144	*	**3**	**+**	**3**	**+**	**3**	**+**	**7**	**+**	_ooooooo	**15**	**+**
144	*	**3**	**+**	**3**	**+**	**3**	**+**	**7**	**+**	oooooooo	**11**	**+**
144	*	**3**	**+**	**3**	**+**	**3**	**+**	**7**	**+**	oooooooo	**12**	**+**
144	*	**3**	**+**	**3**	**+**	**3**	**+**	**7**	**+**	oooooooo	**13**	**+**
correct		**29**		**28**		**28**		**29**			**29**	
missed		0		1		1		0			0	
incorrect		*1*		*1*		*1*		*1*			*1*	

Medium Last Value Predictability												
	L	CE1		CE1		CE3		CE4		SAg CE		
load value	V	C	P	C	P	C	P	C	P	history	C	P
556550514	*	**3**	**+**	**3**	**+**	1		1		_oo_oo_o	3	
1714188861		*3*	-	*3*	-	*2*	-	2		oo_oo_oo	0	
1714188861	*	**2**	**+**	2		0		0		o_oo_oo_	**15**	**+**
1714188861	*	**3**	**+**	**3**	**+**	1		1		_oo_oo_o	4	
556550514		*3*	-	*3*	-	*2*	-	2		oo_oo_oo	0	
556550514	*	**2**	**+**	2		0		0		o_oo_oo_	**15**	**+**
556550514	*	**3**	**+**	**3**	**+**	1		1		_oo_oo_o	5	
1714188861		*3*	-	*3*	-	*2*	-	2		oo_oo_oo	0	
1714188861	*	**2**	**+**	2		0		0		o_oo_oo_	**15**	**+**
556550514		*3*	-	*3*	-	1		1		_oo_oo_o	6	
556550514	*	**2**	**+**	2		0		0		oo_oo_o_	**15**	**+**
556550514	*	**3**	**+**	**3**	**+**	1		1		o_oo_o_o	0	
1714188861		*3*	-	*3*	-	*2*	-	2		_oo_o_oo	0	
1714188861	*	**2**	**+**	2		0		0		oo_o_oo_	**15**	**+**
556550514		*3*	-	*3*	-	1		1		o_o_oo_o	0	
556550514	*	**2**	**+**	2		0		0		_o_oo_o_	**11**	**+**
1714188861		*3*	-	*3*	-	1		1		o_oo_o_o	1	
1361864236		*2*	-	2		0		0		_oo_o_o_	0	
1361128529		1		1		0		0		oo_o_o__	0	
1361128529	*	0		0		0		0		o_o_o___	**15**	**+**
1714188861		1		1		1		1		_o_o___o	0	
1714188861	*	0		0		0		0		o_o___o_	**15**	**+**
1714188861	*	1		1		1		1		_o___o_o	**12**	**+**
556550514		*2*	-	2		*2*	-	2		o___o_oo	0	
556550514	*	1		1		0		0		___o_oo_	**15**	**+**
556550514	*	**2**	**+**	2		1		1		__o_oo_o	**11**	**+**
1714188861		*3*	-	*3*	-	*2*	-	2		_o_oo_oo	0	
1714188861	*	**2**	**+**	2		0		0		o_oo_oo_	**15**	**+**
556550514		*3*	-	*3*	-	1		1		_oo_oo_o	2	
556550514	*	**2**	**+**	2		0		0		oo_oo_o_	**15**	**+**
correct		**13**		**4**		**0**		**0**			**13**	
missed		4		13		17		19			4	
incorrect		*11*		*9*		*6*		*0*			*0*	

Table 5.3: *Bimodal* and *SAg* CE behavior on three *gcc* traces.

The first column in each trace shows the actual load value, the second column (LV) indicates with a star which of the values are last value predictable, the next four columns, CE1 through CE4, show the values of the saturating counters of four *bimodal* CEs as well as the prediction outcome (plus = correct prediction, minus = misprediction, space = no prediction), and the final column (*SAg* CE) shows the history (for better readability, underscores are used to mark correct predictions), the value of the saturating counter corresponding to the given history, and the prediction outcome of a *SAg* CE. At

the bottom of each trace the number of correct predictions, missed opportunities for making a correct prediction, and the number of incorrect predictions are summarized for each individual CE.

The five sample CEs are configured as follows. The *bimodal* CE1 has a top of four, a threshold of two, and a penalty of one. CE2 has a top of four, a threshold of three, and a penalty of one. CE3 has a top of four, a threshold of two, and a penalty of two. Finally, CE4 has a top of eight, a threshold of four, and a penalty of two. The *SAg* CE uses eight-bit histories, a top of sixteen, a threshold of eight, and a penalty of four.

As Table 5.3 illustrates, both the *SAg* and the *bimodal* CE are well suited for predicting highly predictable and highly unpredictable sequences of load values. With mixed predictability, however, the *SAg* CE significantly outperforms the *bimodal* CEs. This is reassuring because, after all, the *SAg* CE has been developed to perform better than the *bimodal* CE in exactly this case. Note that with all three traces, none of the *bimodal* CEs result in more correct predictions, fewer missed opportunities, or fewer incorrect predictions than the *SAg* CE. While this is true for most traces, there are examples in which the *bimodal* CE outperforms the *SAg* CE. A study of such traces revealed that this behavior occurs when the histories of several loads alias detrimentally in the second level of the *SAg* CE, which suggests that in certain cases it may be beneficial to have a hybrid CE that consists of both a *bimodal* and a *SAg* CE and chooses the better one for each load instruction. The analysis of such a CE is left for future work.

To determine the genuine effectiveness of the *SAg* and the *bimodal* CE, speedup measurements are necessary. To obtain these results, the five value predictors from Section 4.3 were outfitted with both kinds of CEs. Based on previous studies [BuZo98b], a history length of ten bits is used with the *SAg* CEs and the top values for the saturating counters are sixteen for refetch recovery and eight with re-execute. A global search was performed to obtain the optimal threshold and penalty values for each predictor and CE

pair. The result of this search is summarized in Table 5.4.

		bimodal confidence estim.			SAg confidence estimator			
		cntr top	threshold	penalty	hist bits	cntr top	threshold	penalty
re-fetch	FCM	16	13	11	10	16	15	11
	L4V	16	15	13	10	16	15	8
	LV	16	10	15	10	16	13	5
	Reg	16	10	7	10	16	15	7
	St2d	16	12	12	10	16	12	5
re-execute	FCM	8	7	3	10	8	6	3
	L4V	8	6	3	10	8	7	3
	LV	8	5	1	10	8	5	2
	Reg	8	2	2	10	8	4	1
	St2d	8	5	1	10	8	5	1

Table 5.4: Predictor configurations yielding the highest mean speedup.

Note that the penalties yielding the highest performance with a re-execute misprediction recovery mechanism are quite low in comparison with those for re-fetch, even when accounting for the wider re-fetch counters. This is a direct reflection of the lower misprediction cycle-penalty with re-execute.

The speedups delivered by the twenty combinations of predictors, CEs, and recovery mechanisms are shown in Figure 5.3 and Figure 5.4. The former shows the re-fetch speedups and the latter the re-execute speedups. Each predictor comprises a total of 2048 lines divided into four banks. Since the predictor sizes vary greatly, the given results should only be used for intra-predictor comparisons between the two kinds of CEs.

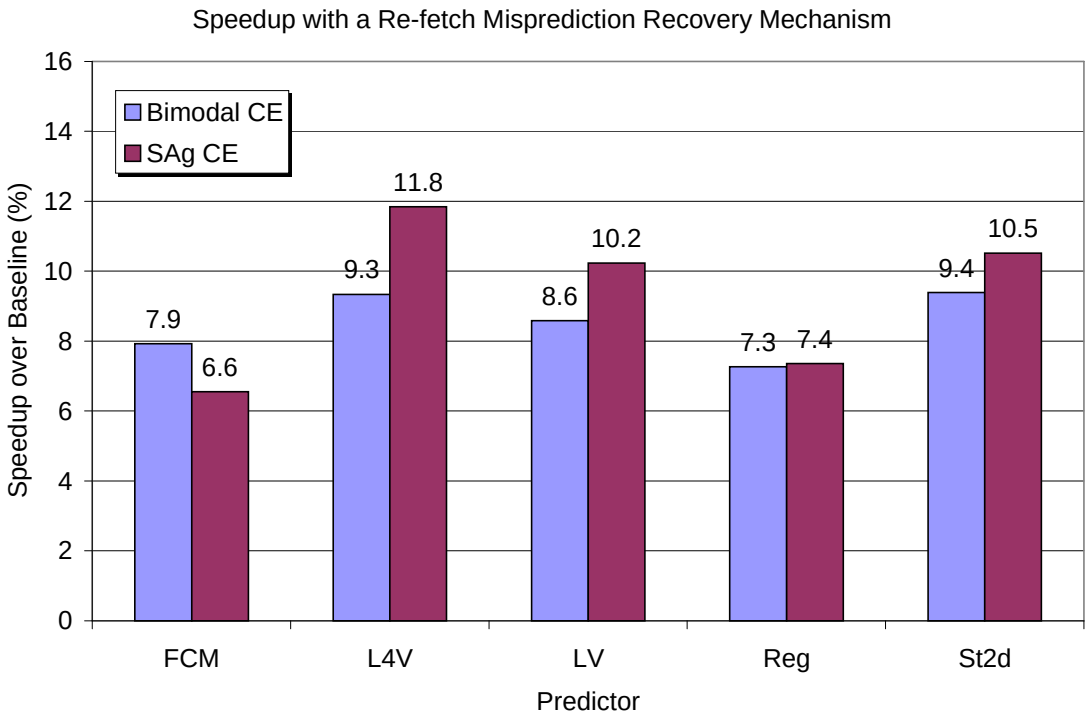

Figure 5.3: Re-fetch speedup comparison between *Bimodal* and *SAg* CEs.

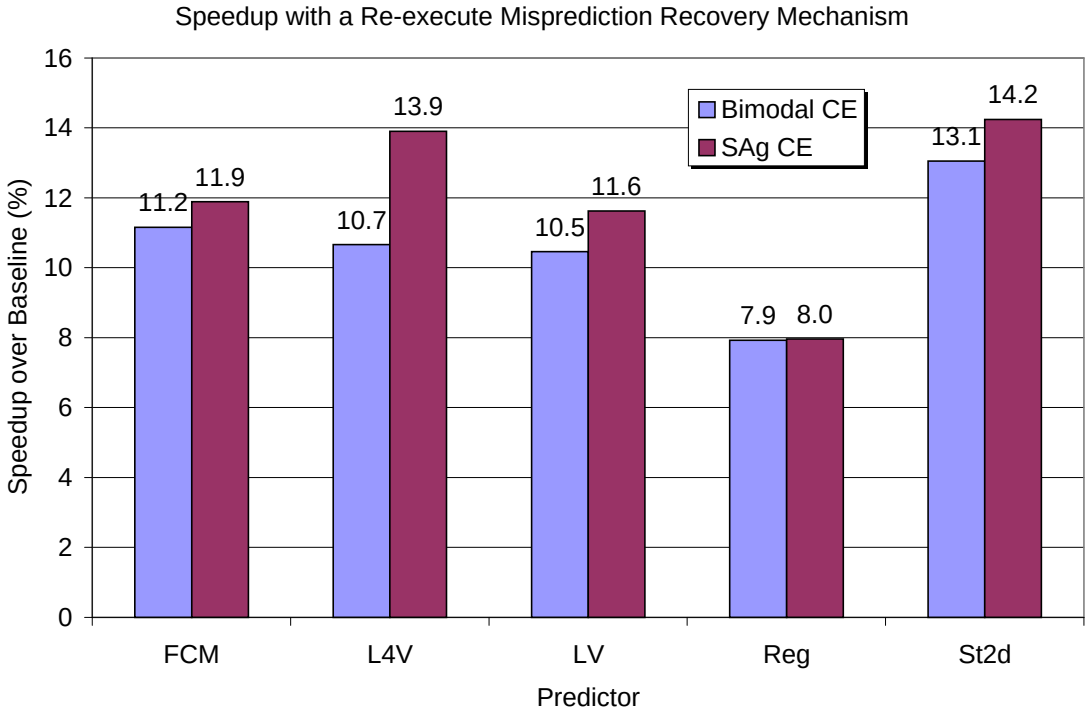

Figure 5.4: Re-execute speedup comparison of *Bimodal* and *SAg* CEs.

With a re-fetch misprediction recovery mechanism, four of the five predictors perform better with a *SAg* CE than with a *bimodal* CE. The L4V, St2d and, as expected, the LV predictors' speedups are considerably higher with a *SAg* CE. Surprisingly, the Reg predictor does not seem to benefit much from the more complex CE at all.

The FCM predictor actually performs substantially worse with a *SAg* CE than with a *bimodal* CE. This unexpected result is caused by the aforementioned detrimental aliasing in the second level of the *SAg* CE. As Figure 5.4 shows, aliasing is less of a problem (but still present) with the FCM predictor when re-execute recovery is used.

With a re-execute misprediction recovery mechanism, all five predictors perform better with a *SAg* CE than with a *bimodal* CE. However, again the Reg predictor does not seem to benefit much from the *SAg* CE. It appears that register predictable loads are either highly predictable or not predictable at all and that there is not much alternating register predictability. Clearly, the *bimodal* CE is the CE of choice for the Reg predictor.

While the re-execute speedups are higher than the re-fetch speedups, the delivered speedup is considerable for all five predictors even with re-fetch recovery, in particular in comparison with the speedups achieved without CEs, as shown in Figure 4.8. I conclude that confidence estimators make load value predictors somewhat larger and more complex but definitely appear to be worthwhile having. In fact, even the best predictor without a CE does not come close to the speedup of the worst studied predictor with a CE.

5.4.1 The L4V Selector

Interestingly, the L4V predictor benefits the most from having a *SAg* CE over having a *bimodal* CE with both misprediction recovery mechanisms. To explain this fact, it is necessary to understand the implementation of the selection mechanism in the L4V predictor.

The L4V comprises four replicated LV predictors with their CEs. Each component of the L4V operates independently, meaning that they each make a value prediction and a confidence estimation in parallel. Whichever component reports the highest confidence is chosen to make the actual load value prediction if the confidence is also above the threshold. In case of a tie the component with the youngest value is selected. The implementation of the select function from Section 4.3.4 can now be given.

$$\begin{aligned}\text{select}(val1, val2, val3, val4) = {} & val1 \text{ if } val1_{conf} = \max(val1_{conf}, val2_{conf}, val3_{conf}, val4_{conf}),\\ & val2 \text{ if } val2_{conf} = \max(val1_{conf}, val2_{conf}, val3_{conf}, val4_{conf}),\\ & val3 \text{ if } val3_{conf} = \max(val1_{conf}, val2_{conf}, val3_{conf}, val4_{conf}),\\ & val4 \text{ otherwise;}\end{aligned}$$

The confidence information is therefore not only used to determine whether a value prediction should be allowed but also which component of the L4V predictor to select. Hence, the fact that the L4V benefits more from the *SAg* CE relative to the *bimodal* CE than the LV predictor implies that the *SAg* CE represents a better selector than the *bimodal* CE.

5.4.2 Other Performance Metrics

Often, metrics other than the speedup are used to evaluate the effectiveness of CEs and value predictors (see Section 3.4). For example, the percentage of correct predictions, incorrect predictions, and inhibited predictions is of interest for load value predictors with CEs. Figure 5.5 shows this classification for the twenty predictor configurations under discussion.

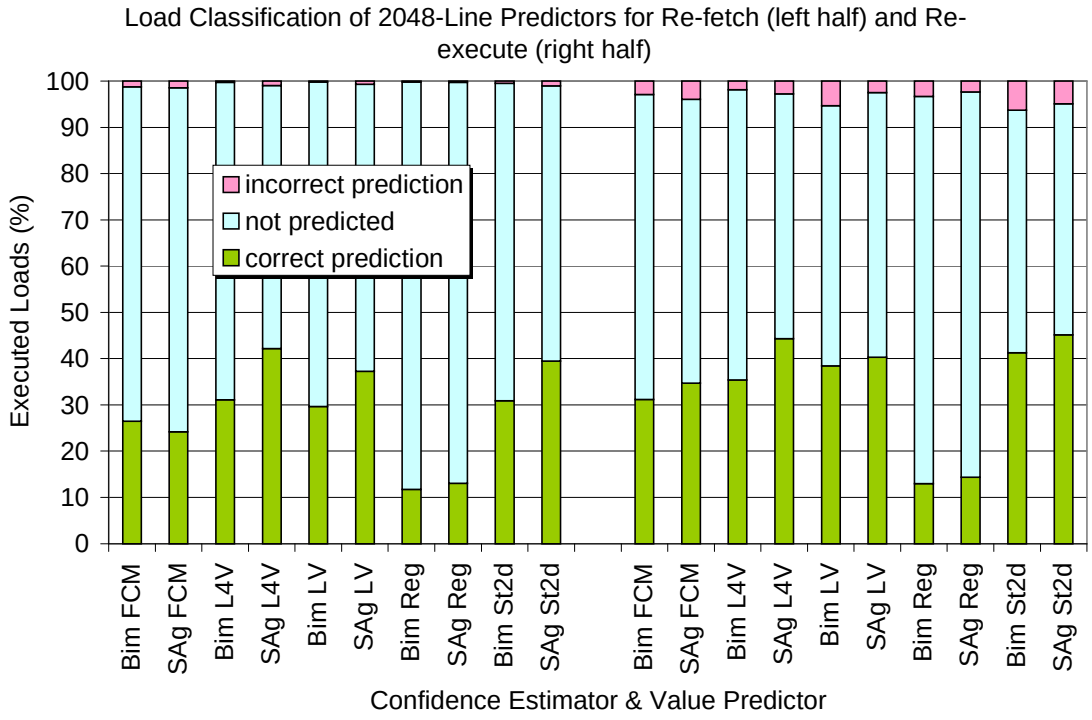

Figure 5.5: Load classification of *Bimodal* and *SAg* confidence estimators.

It is quite evident in Figure 5.5 that fewer overall predictions are attempted with re-fetch than with re-execute. While this conservativism significantly reduces the number of incorrect predictions, it also lowers the number of correct predictions. However, this trade-off must be beneficial in the modeled CPU since the configurations used yield the highest speedups.

Note that none of the predictors predict more than 45% of the dynamically executed loads correctly. In particular, the Reg predictor only correctly predicts between eleven and fifteen percent of all the loads. In spite of this very low prediction rate, the Reg predictor yields a respectable speedup. One can speculate from this that the loads the Reg predictor is able to predict are more important than the loads the remaining four predictors can predict. As it turns out, this is indeed so. The loads predicted by Reg have a substantially longer average latency than the ones predicted by the other four predictors, as the shaded columns in Table 5.5 illustrate. For example, the loads predicted by the Reg predictor have an average latency of over twenty cycles both with re-fetch and re-execute recovery whereas the St2d's loads only have a latency of 12.5 cycles for re-fetch and about fifteen for re-execute.

	re-fetch		re-execute	
	latency	usage	latency	usage
Bim FCM	15.0	3.3	17.5	3.8
SAg FCM	14.6	3.8	16.6	4.0
Bim L4V	14.9	4.5	15.3	5.8
SAg L4V	14.6	4.9	16.8	5.7
Bim LV	15.3	4.7	16.6	5.2
SAg LV	15.2	5.0	17.2	5.7
Bim Reg	21.2	3.3	20.7	4.7
SAg Reg	20.4	3.9	20.2	4.9
Bim St2d	12.1	3.4	14.8	3.2
SAg St2d	12.7	3.0	15.4	3.2
average	15.6	4.0	17.1	4.6

Table 5.5: Latency and cycles to first usage of the predicted load values.

Evidently, the number of correct predictions, the number of incorrect predictions, and the prediction rate (the sum of the two) do not adequately pre-

dict the delivered performance of a CE. Rather, it appears that the latency of the predicted loads also has to be taken into account.

Table 5.6 demonstrates two even more striking examples of metric-anomalies. The first two lines show that while the *bimodal* FCM makes fewer correct predictions, more incorrect predictions, fewer prediction attempts, and has a lower prediction rate and a lower accuracy than the *bimodal* L4V with re-execute, the FCM still outperforms the L4V speedup-wise. To ensure that this result is not an artifact of the averaging of the speedups, the table shows the harmonic, geometric, and arithmetic mean speedup as well as the average IPC (instructions per cycle) improvement of the eight SPECint95 programs. All four ways of averaging the measured speedups yield the same result, i.e., the FCM performs better than the L4V. Again, the higher average latency (Table 5.5) of the loads predicted by the FCM predictor appears to offer at least one explanation. The second example in Table 5.6 illustrates another possible reason.

	% correct predictions	% no predictions	% wrong predictions	prediction rate (%)	accuracy (%)	mean speedup over baseline (%) harmonic	geometric	arithmetic	IPC
Bim FCM	31.15	65.93	2.93	34.08	91.40	11.16	13.63	16.66	13.03
Bim L4V	35.37	62.73	1.90	37.27	94.90	10.66	12.34	14.50	11.58
SAg FCM	34.71	61.34	3.95	38.66	89.78	11.88	14.89	18.55	14.60
SAg LV	40.28	57.26	2.46	42.74	94.24	11.63	13.36	15.59	12.47

Table 5.6: Various metrics showing anomaly.

The second set of two lines in Table 5.6 shows that according to all non-speedup metrics the *SAg* FCM should perform worse than the *SAg* LV but again all the shown re-execute speedup averages are in disagreement. This time even the load latency is in favor of the LV, meaning that there has to be at least one other as of yet unaccounted for influence on the CE performance.

The non-shaded columns of Table 5.5 offer a possible explanation. The average time to the first usage of a predicted load value is much lower for the

FCM (4.0 cycles) than it is for the LV (5.7 cycles), meaning that the FCM's predictions are needed sooner by the CPU and are therefore more important than the LV's. Again, it looks like the time to the first use of a predicted load value needs to be accounted for to properly establish a CE's performance.

Another issue with non-speedup metrics is the time (or physical location) of the actual measurement. Optimizing predictors for speedup implies optimizing the performance at instruction commit. The interaction between the CPU and the predictor, however, take place at the time of prediction and then again at the time of update, possibly long before the time of commit. This discrepancy may be an issue because, for instance, the accuracy with which wrong path instructions are predicted is most likely less important than the accuracy of correct path instructions. Hence, a high overall accuracy measured at predict or update may not be representative of the predictor's performance since it makes no statement about the prediction accuracy of the instructions that are actually retired. The ratio of total predicted loads over committed value-predicted loads is just under 1.5, indicating that a substantial number of predictions exist that probably have little impact on the overall performance. To account for any effects this might have, out-of-order and wrong-path updates of the predictor may have to be accurately modeled and non-speedup events should be sampled in the commit stage of the CPU and not at the time of prediction or update.

5.5 Summary

This chapter introduces confidence estimators, which are an essential part in every load value predictor, as performance numbers illustrate.

First, the simple but effective *bimodal* confidence estimator is presented and its operation is described. A behavior study revealed a deficiency of this CE on sequences of load values that frequently change from being predictable to being unpredictable and vice-versa.

To alleviate this problem, the more complex *SAg* CE is derived. Speedup results show that it performs better in connection with most load value predictors than the *bimodal* CE. Furthermore, there is strong evidence that the *SAg* CE represents a better selector than the *bimodal* CE in hybrid predictors.

A study of the expressiveness of non-speedup-based metrics concludes this chapter. Interestingly, all the simple metrics that are discussed appear to be misleading in some cases, which is why speedup numbers are used almost exclusively in this dissertation for performance evaluation purposes.

Chapter 6

Predictor Banking

This chapter discusses predictor banking, a technique used to enable multiple predictor accesses (predictions and updates) per cycle. Several measurements show that a banked predictor design is necessary but also sufficient for good load value predictor performance.

6.1 The Need for Banking

As discussed in Section 3.2.3, about 23.3 percent of the committed instructions in the SPECint95 benchmark programs are loads. With an average IPC (instructions per cycle) of 1.677, this results in roughly one executed load instruction every 2.5 cycles. Since each load accesses the predictor twice, once to request a prediction and once to update the predictor, the predictor is accessed once every 1.25 cycles. When also accounting for wrong-path loads and loads that are re-executed, the number of predictor accesses increases to 0.962 per cycle on average. However, since prediction and update requests are not evenly distributed over time, it frequently happens that more than one access per cycle is needed.

Moreover, load value predictors improve the CPU throughput (performance), which in turn increases the pressure on the predictor because there is less time between the execution of consecutive load instructions. The column “accesses per cycle“ in Table 12.1 in Appendix B shows the average number of predictor accesses for various predictor configurations. As can be seen, with some of the predictors even the average number of accesses ex-

ceeds one per cycle, clearly demonstrating the need for multi-access support. Limiting the predictor to one prediction or update per cycle severely hampers the performance (see Section 6.3), which also shows the importance of an architecture that supports more than one access per cycle.

6.2 Bank Architecture

One approach to enable multiple accesses per cycle is to break the predictor up into multiple predictor banks [GaMe98]. In such a predictor, each bank comprises a small, identical load value predictor. There is no communication between the banks, making it possible to operate them independently and in parallel. While each bank by itself is still only able to handle one prediction or update per cycle, taken together the *n* banks support up to *n* accesses per cycle.

The number of banks needed depends on the maximum number of accesses that the predictor should be able to handle per cycle. Since the CPU I use can issue up to four load instructions per cycle, a load value predictor with four banks is probably necessary. The results in the following section verify this assumption.

While most microprocessors currently only support one issued load per cycle (because they only have one load/store unit), this issue-rate is likely to increase in the near future. With every fifth instruction being a load, eight-way superscalar CPUs already take a considerable performance hit when only allowing one load per cycle.

I decided to allow up to four loads to issue in the simulated CPU to illustrate that predicting and/or updating multiple loads per cycle is straightforward in load value predictors. Note that the baseline CPU, which does not contain a load value predictor, is also able to issue up to four loads per cycle. The four-wide load issue width results in a high-performing baseline CPU, which makes it harder for a load value predictor to be effective and ensures that no

performance improvements are attributed to the load value predictors that are actually an artifact of a limited load issue width. Limiting the load issue width to one decreases the baseline processor's performance more than the performance of the CPU with a load value predictor. As a consequence, the speedup (i.e., the performance of the load value predictor) would appear to be higher.

I believe that if I can show load value predictors to be effective under the more demanding conditions of a four-wide baseline processor, there is a good chance that load value predictors will be included in future microprocessors and be effective for years to come.

Since the simulated processor I use mimics an Alpha 21264, it fetches naturally aligned instructions. Consequently, any set of up to four loads that can be fetched or issued during the same cycle can only contain loads whose addresses (PC values) differ in the two least significant bits (that are not always zero). Using these two bits to determine which bank a load should be handled by guarantees that there is never a bank conflict between issued loads and results in an interleaved bank design as illustrated in Figure 6.1.

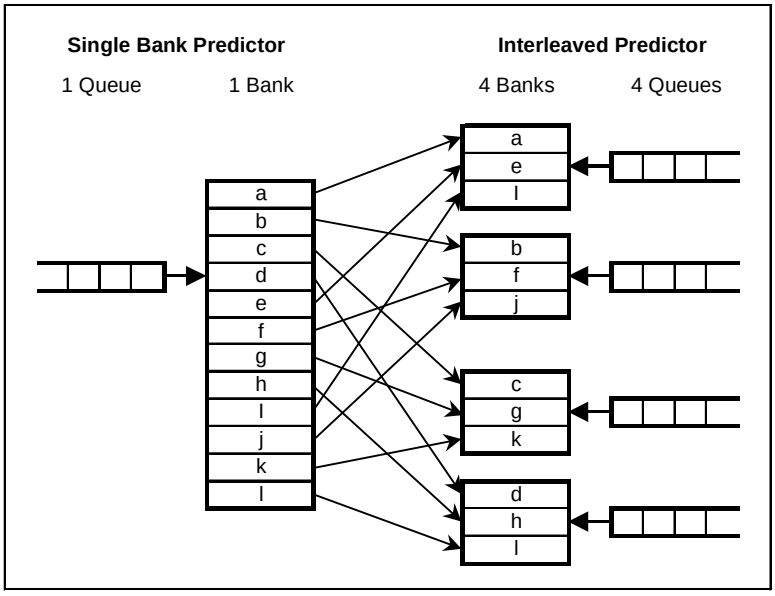

Figure 6.1: Line correspondence of single-bank and interleaved predictor.

Because the address of an instruction never changes during program execution, a given load will always be handled by the same predictor bank. With an interleaved banking scheme, all the load instructions that the modeled CPU can possibly fetch during the same cycle not only go to distinct banks but the first of the four instructions will always be handled by the first predictor bank (if it is a load), the second instruction by the second bank and so forth. Hence, neither arbitration nor rotation logic is necessary for predictions. For CPUs that fetch contiguous but not necessarily naturally aligned sets of instructions, the relative position of the instructions within the fetch block may have to be rotated before accessing the predictor banks. There should be enough time available to perform this rotation during the decode stage, making the interleaved banking scheme applicable to a broad range of processors. Gabbay and Mendelson describe a more complex predictor banking scheme for CPUs with trace-caches that do not necessarily fetch contiguous instructions [GaMe98].

Unfortunately, predictor updates also take time and keep the predictor bank that is being updated busy for one cycle during which it is not available for making a prediction. Since it is vital for good performance that the predicted load values be available as soon as possible, updates should be given a lower priority than predictions. Hence, updates are only allowed during cycles when the respective bank is not making a prediction, i.e., when it is idle.

To avoid dropping (i.e., losing) updates whenever the bank is busy, which would considerably decrease the performance of the predictor, updates are temporarily stored in a FIFO (first-in first-out) queue. Each predictor bank contains one such queue that can accept one update per cycle. Updates are only dropped if the queue is full. Whenever a queue is not empty and the corresponding predictor bank is idle, the queue issues updates at a rate of one per cycle. The following section shows that sixteen-entry queues are sufficient to essentially avoid dropping any updates.

6.3 Bank Performance

Figure 6.2 shows the speedup delivered by the five basic load value predictors from Section 4.3 with one, two, and four (interleaved) predictor banks as well as with four banks that support an infinite number of accesses per cycle (denoted as “unlimited”). Each predictor has a total of 2048 lines that are equally distributed among the banks, meaning that the total predictor size is roughly the same for the four configurations. Nevertheless, the two and in particular the four bank predictors require a little more state than their single bank counterparts because each bank requires its own sixteen-entry update queue and second level of the *SAg* confidence estimator in order to be independent of the other banks. The CEs are configured with the parameters shown in Table 5.4.

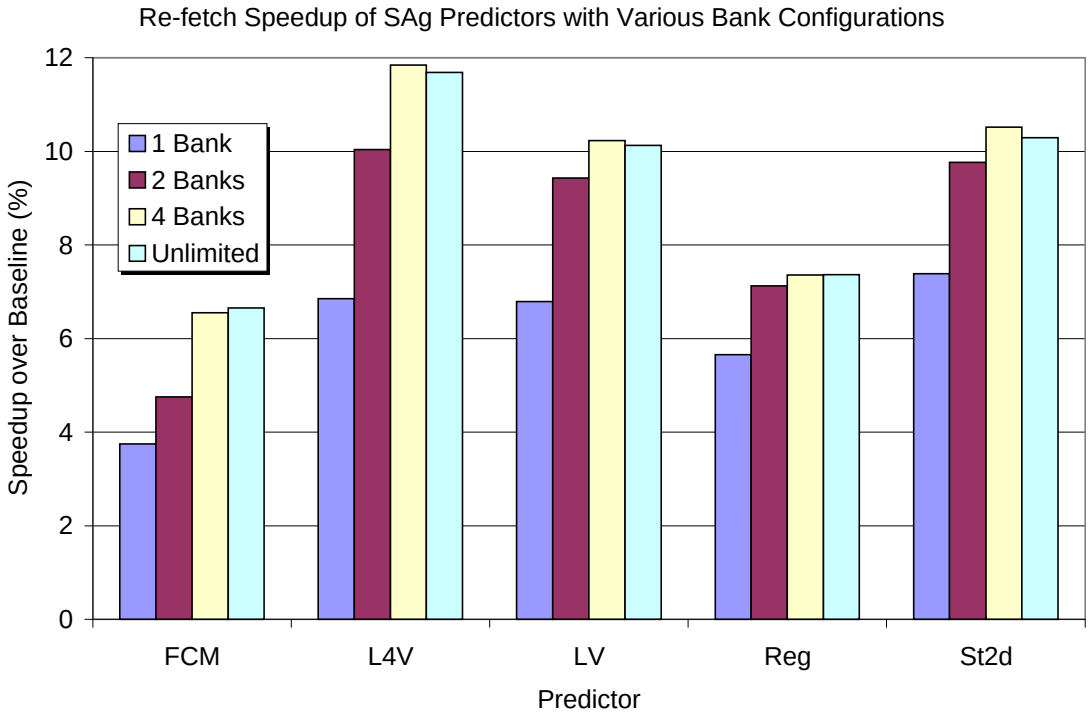

Figure 6.2: Re-fetch speedup of differently banked *SAg* predictors.

Figure 6.2 illustrates the performance of the five predictors with *SAg* CEs and a re-fetch misprediction recovery mechanism. The results for the predictors with *bimodal* CEs and re-fetch, with *bimodal* CEs and re-execute, and with *SAg* CEs and a re-execute exhibit the same general trends and can be found in Appendix B.

Clearly, the single-bank predictors perform substantially worse than their four-banked counterparts in all cases. On average, the roughly same sized four-bank predictors outperform the one-bank predictors by 42.7%. The two-bank predictors outperform the one-bank predictors by 26.4% on average but underperform their four-banked counterparts in all cases.

Interestingly, the four-banked predictors sometimes perform slightly better than the same predictors with four banks that support an unlimited number of accesses per cycle. Since the predictors are structurally identical and only differ in their timing behavior, this apparent paradox must be explainable by the delayed updates. The predictions cannot be the reason because their timing behavior is the same in both cases since the four banks never drop a prediction and predictions take precedence over updates. Hence, predictions happen at the same time in both the four-banked and the unlimited predictors.

The updates, however, take place later in the implementable four-bank predictors than in the unlimited predictors. At first sight it seems illogical that a predictor that is more likely to contain out-of-date information would outperform a more up-to-date predictor. While I do not have proof for this, I believe that the outdated value information indeed decreases the performance but the outdated confidence information is actually helpful for bridging phase transitions by providing some hysteresis, which could explain the slightly improved performance. A detailed analysis of this phenomenon and possible exploitation thereof is left for future work.

Figure 6.2 (and the results shown in Appendix B) provides strong evidence that a processor should have one predictor bank for every load that it

can issue per cycle. In our case, the four-bank implementations outperform their counterparts with fewer banks, which are scaled to the same total size. At the same time, the four-banked predictors yield the same performance as the unlimited predictors, meaning that a simple interleaved banking scheme is sufficient to reap the full potential of the predictor.

Because predictions take precedence over updates and because no more than four predictions need to be made per cycle, no predictions are ever dropped in the four-bank case. With only two banks, nineteen percent of the requested predictions have to be dropped on average because the required bank is already busy making another prediction. With one bank, 50.4% of the predictions cannot be made due to a busy predictor. Table 12.1 in Appendix B shows the percentage of dropped predictions for different numbers of banks and a variety of load value predictors.

Delaying updates by a few cycles (in the update queue) and dropping updates when the queue is full appears not to impact the performance at all. The sixteen-entry update queues are large enough so that on average only 0.026% of all the updates have to be dropped due to a full queue in the four-bank case. With two banks, the queues are full 2.1% of the time and with just one bank 23.9% of the updates have to be dropped on average. The drop-rates of the individual predictors are listed in Appendix B. The investigation of the performance of shorter update-queues is left for future work.

Table 12.1 in Appendix B lists relevant bank information for the five basic predictors with one, two, and four banks for all combinations of re-fetch and re-execute as well as *bimodal* and *SAg* CEs. The information in the table includes the average number of load instructions that are retired per cycle, the average number of load value predictor accesses per cycle, the average number of references to the load value predictor per cycle (i.e., the number of prediction requests), the average number of updates per cycle, the percentage of prediction requests that have to be dropped due to a busy bank, and the percentage of dropped updates due to a full update queue.

As the data in the table shows, the number of prediction requests per cycle is often higher than the number of updates because of wrong-path load instructions that request a prediction but are cancelled before they can update the predictor. However, in a few cases there are more updates than predictions. This only happens with re-execute, though, because re-executed load instructions update the predictor multiple times.

Note also that for both re-fetch and re-execute, the number of predictor accesses is on average about 2.23 times higher than the number of committed load instructions. Assuming exactly two predictor accesses per load, we find that in addition to the committed loads, another 11.5% of load instructions affect the predictor that stem from wrong path executions.

6.4 Bank Usage

The utilization of the four predictor banks is almost identical for all the tested predictor configurations. The first bank is accessed 27.6%±0.2% of the time (depending on the predictor), the second bank 21.8%±0.1%, the third bank 29.3%±0.2%, and the fourth bank 21.3%±0.2% of the time. This result is hardly surprising since the utilization is a function of the distribution of load instructions in the fetch blocks. The observed percentages show that this distribution is quite uniform and not biased by the code scheduler. Hence, it appears that banking does not result in uneven utilization.

6.5 Summary

The performance numbers in this chapter show that in order to be effective, a load value predictor has to support multiple accesses per cycle in connection with a CPU that has a load-issue-width greater than one. Splitting a predictor into multiple interleaved banks is a straightforward approach and

provides the needed support for multiple simultaneous predictor accesses. The performance results show that such a banking scheme can deliver the same speedup as a predictor that supports an unlimited number of accesses per cycle. Unless otherwise noted, all the predictors in this dissertation are split into four predictor banks.

Chapter 7

Improving Predictor Utilization

This chapter investigates the utilization of load value predictors and suggests an alternative predictor design that performs better due to improved and more balanced hardware utilization.

Furthermore, speedup results for individual programs and not only averages over the whole benchmark suite are presented. A sensitivity analysis of several last four value predictor parameters concludes this chapter.

7.1 Line Utilization

While every line in a load value predictor requires the same amount of state to store information, not every line is used equally frequently. In fact, even in relatively small predictors most of the lines are seldom utilized, as the quantile information from Section 3.2.2 illustrates. As discussed, only a small percentage of the load instructions contained in a binary contributes most of the dynamically executed loads. For example, Table 3.3 shows that on average only 36.6% of the load sites (the static load instructions in the binary) are visited at all during execution, 3.5% of the load sites contribute ninety percent of the executed loads, and less than one percent of the load sites contributes over half of the dynamically executed loads.

Clearly, only a small percentage of the load sites is responsible for most of the executed loads. As a consequence, only a few lines in the load value predictor are accessed most of the time and thus have to handle most of the predictions and updates.

To find out what number of predictor lines is really needed to handle a given percentage of executed loads in absolute terms, the number of load sites that account for the given percentages of executed loads is shown in Figure 7.1 for each of the eight SPECint95 programs individually. Note that the scale on the y-axis is logarithmic.

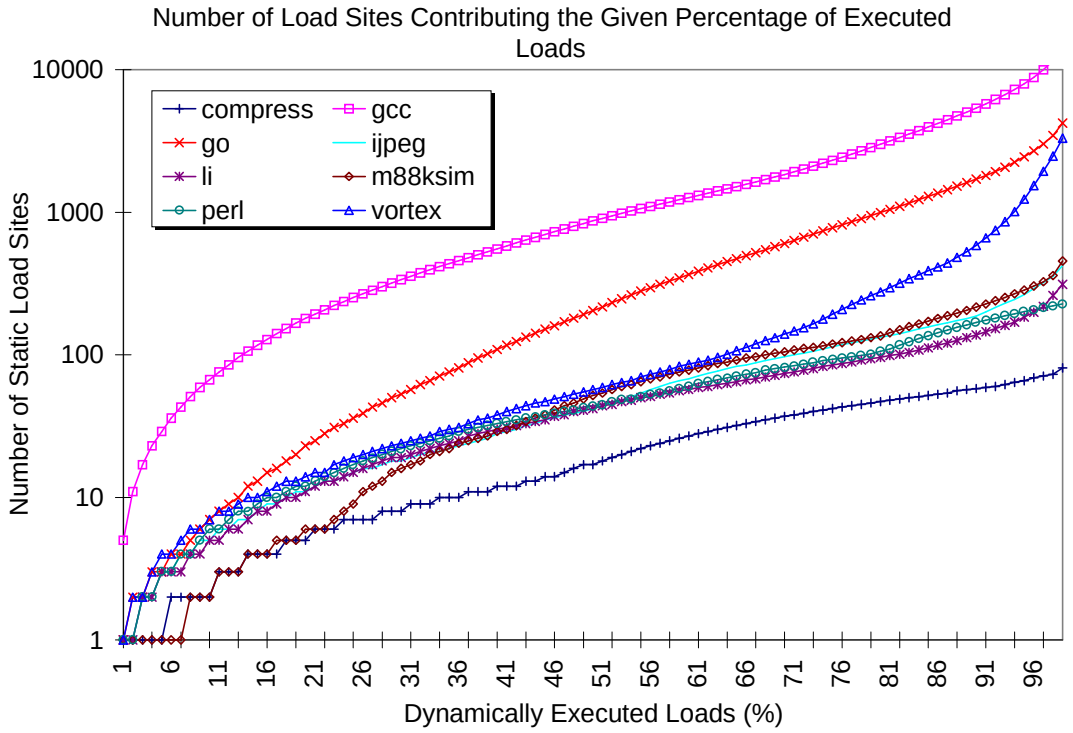

Figure 7.1: Absolute quantile numbers for the eight SPECint95 programs.

With the exception of *gcc*, which is by far the largest of the eight programs, ten load sites and therefore ten predictor lines handle almost fifteen percent of the executed loads. Except for *gcc* and *go*, fewer than sixty predictor lines are needed to cover over half of all the predictions and updates. The program *go* requires about two hundred lines and *gcc* about nine hundred predictor lines to handle half of all the executed loads. Again with the exception of *gcc* and *go*, fewer than six-hundred predictor lines are necessary to handle ninety percent of the predictor traffic.

As these numbers illustrate, over half of all the predictor lines in a 2048-line predictor are scarcely used when running the SPECint95 programs. This observation prompted me to investigate alternative predictor designs to improve the utilization of the hardware, hoping that better utilization will also result in higher performance.

One generic approach to increase the usage of the available real-estate is to take away hardware from the infrequently executed loads and "give" it to the frequently executed load instructions. Doing so reduces the predictor's capability to predict infrequently executed loads, but at the same time it should increase the prediction accuracy and the number of times the frequently executed loads can be predicted. Due to the widely varying execution frequency of loads, missing infrequently executed loads and better predicting frequently executed loads is likely to be a beneficial trade-off.

7.2 Trading off Height for Width

An increase in prediction accuracy and in particular in the absolute number of correct predictions can be achieved by increasing the amount of information available for making predictions. For example, instead of only retaining the last seen load value, one could store the last *n* load values in every predictor line. Of course, this would significantly increase the size of the predictor, which is why the number of predictor lines has to be reduced by a factor of *n* if the overall predictor size is to be maintained.

Reducing the number of predictor lines and increasing the amount of information stored in each line has the desired effect of retaining more information about frequently executed load instructions while expelling infrequently executed loads from the predictor (because of the smaller number of predictor lines).

A predictor that stores the last *n* load values is called a *last n value predictor*. The *last four value predictor* from Section 4.3.4 is an example of such a

predictor. As explained in Section 5.4.1, the confidence estimators in each of the *n* predictor components can be used to select one of the *n* components for making the next prediction by "measuring" which of the *n* values is the most likely to result in a correct prediction.

The architecture and operation of a last *n* value predictor is identical to that of the last four value predictor except that there may be fewer or more than four values per predictor line. In fact, the *last value predictor* (Section 4.3.1) is one extreme of the more general last *n* value predictor (*n* = 1).

The number *n* is often referred to as the width of a last *n* value predictor. The wider such a predictor is, the fewer lines it can have for a given predictor size. Hence, there exists a trade-off between the predictor's height (the number of lines) and its width.

Based on the quantile numbers from the previous section, it is likely that reducing the predictor height and increasing its width is beneficial until the predictor becomes so short that it cannot hold the frequently executed load instructions anymore or until retaining additional values in the predictor lines no longer results in better predictions. Performance studies are therefore necessary to find the optimal width and height of a last *n* value predictor for a given predictor size and workload.

7.3 *SAg* L4V Predictor Design and Performance

The previous section established that a load value predictor's height ought to be tall enough (i.e., have a sufficiently large number of lines) to accommodate the load instruction working set size. If the predictor is too short, some frequently executed load instructions will have to share a predictor slot, which almost always results in detrimental aliasing. Predictors that are too tall, on the other hand, underutilize their hardware. The optimal predictor width therefore depends on the working set size of the programs and the available predictor real-estate.

To better evaluate the trade-off between predictor height and width, Figure 7.2 and Figure 7.3 are presented. They show the mean speedup of a last one, two, four, eight, and sixteen value predictor with re-fetch and a re-execute recovery, respectively. Three speedup numbers are given for each predictor. The first one shows the speedup for a predictor with a total capacity of 512 load values, the second one for a predictor size of 2048 values, and the last one for a predictor with 8192 values. For each predictor size and width, the performance obtained with the best threshold and penalty values is shown. All the predictors are partially tagged and contain *SAg* confidence estimators with ten-bit histories. The counter top is sixteen with re-fetch and eight with re-execute. The CE is also used as selector.

Figure 7.2 shows that for small predictors with only four kilobytes of state for storing values (512 values), a width of one results in the highest speedup. Storing two values per line and halving the number of predictor lines yields less speedup because there are not enough lines left for the frequently executed loads in the SPECint95 programs, which results in detrimental aliasing and thus lower performance. The shorter the predictor the more pronounced the aliasing is, hence the continuous decrease in speedup as the predictors become wider.

With sixteen kilobytes of state (2048 values), a width of four results in the highest speedup and the detrimental aliasing only sets in above four entries per predictor line. When the predictor size is increased even further (to sixty-four kilobytes or 8192 values of total storage), the best width turns out to be eight. Only at a width of sixteen does the performance decrease again.

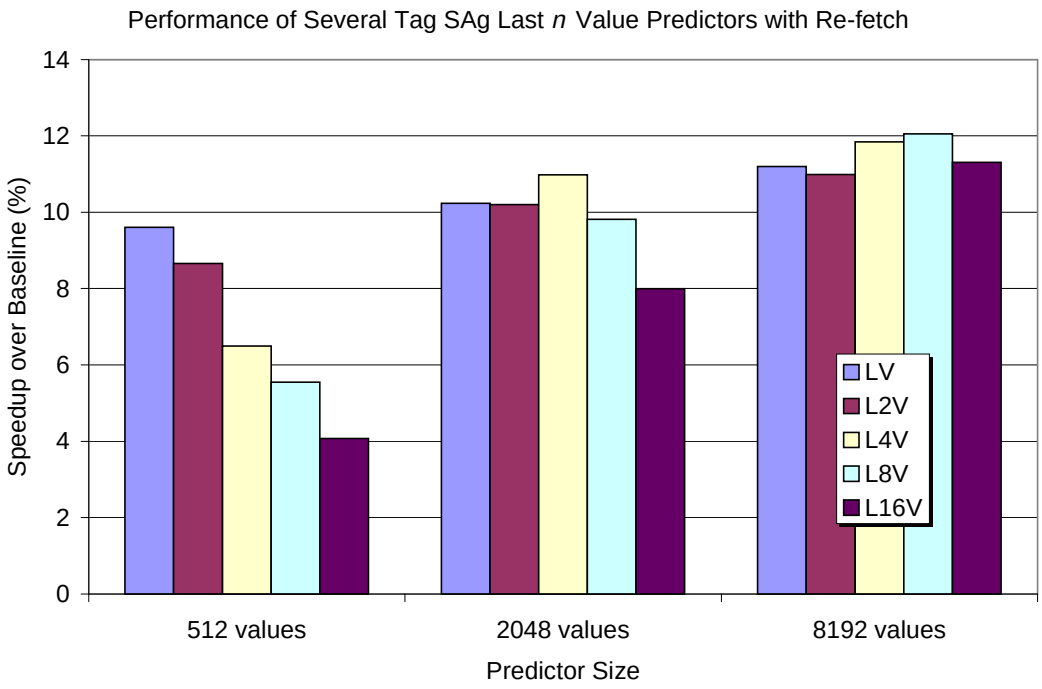

Figure 7.2: Re-fetch speedup of three sizes of last *n* value predictors.

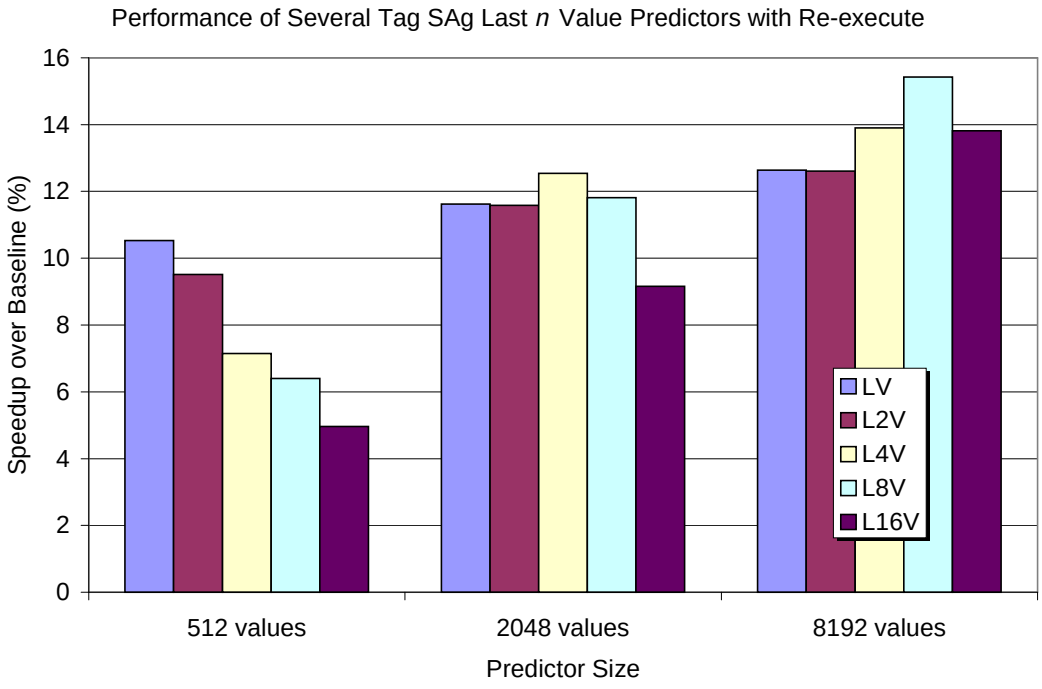

Figure 7.3: Re-execute speedup of three sizes of last *n* value predictors.

Figure 7.3 is identical to Figure 7.2 except that the misprediction recovery mechanism used is re-execute instead of re-fetch. The best performing predictor widths are exactly the same for the three sizes and are even more prominent.

Interestingly, the last two value predictor performs worse than the last value predictor in all presented cases. Since the last two value predictor has a selector and the last value predictor does not, it appears that the loss due to imperfect selections is slightly larger than the benefit of having the two last values available for making predictions. Note, however, that the last two value predictor outperforms the last value predictor when both are given the same number of predictor lines (i.e., the LV predictor is half the size of the L2V predictor) and that the last two value predictor does outperform the same-sized last value predictor for sizes above the depicted 8192-entry predictors.

In general, and as expected, the optimal predictor width increases as the predictors become larger (for a fixed workload). Because of the highly skewed distribution of the execution frequency among load instructions, already relatively small load value predictors benefit from an increase in width even at the cost of a decreased height. For example, both with re-fetch and re-execute, the sixteen kilobyte last four value predictor outperforms the other last *n* value predictors of the same size. Consequently, a width of four and a height of 512 represents the optimal width over height ratio for this workload and predictor size.

7.4 *SAg* L4V Predictor Potential

In brief, the four-banked, 2048-entry eight-bit partially tagged *SAg* last four value predictor's average accuracy over SPECint95 measured in the CPU's commit stage is 97.6% using re-fetch with a counter top of sixteen, a threshold of fourteen, and a penalty of eleven. On average, 32.6% of the commit-

ted load instructions are predicted with the correct value and 0.8% with an incorrect value. This results in a harmonic mean speedup of 11.0% relative to the same CPU without the load value predictor.

With re-execute, a counter top of eight, a threshold of seven, and a penalty of four, the average accuracy of the predicted load instructions that are committed is 94.2%. 34.6% of the load instructions are correctly predicted on average and 2.1% are incorrectly predicted. The resulting harmonic mean speedup is 12.5%.

7.4.1 Comparison with Oracles

To better understand the potential that lies in load value prediction and to see how much of this potential the last four value predictor can reap, I modified the simulator to provide various degrees of perfect knowledge to the load value predictor, i.e., to include oracles that can make perfect predictions.

The first predictor (*no-oracle*) represents the Tag *SAg* L4V predictor in its conventional and implementable form. It has a capacity of 2048 values and does not contain an oracle.

The first oracle (*ce-oracle*) represents the same predictor except it incorporates a perfect confidence estimator. Because the confidence information is always correct, no incorrect predictions are made (they are all inhibited) and the predictor always makes a prediction if the (imperfectly) selected component contains the correct value.

The next oracle (*ce/sel-oracle*) improves on the first one by also including a perfect selector. This means that the oracle not only always makes a prediction if the correct value is available and never makes a prediction otherwise, but also that it chooses the component that will make a correct prediction if such a component exists. Hence, if any component in the predictor can make a correct prediction, it is selected and a prediction is made, otherwise no prediction is attempted. This oracle also never causes a mispredic-

tion.

The final oracle (*all-oracle*) predicts all executed load instructions correctly. The *all(tag)-oracle* does the same except it only attempts a prediction if there is a tag-match in the 512-line predictor. Again, there are no mispredictions. As opposed to all the other oracles, the *all-oracle* never decides not to make a prediction.

Figure 7.4 shows the speedups of the oracle-less L4V predictor and the four oracles with a re-fetch and a re-execute misprediction recovery mechanism.

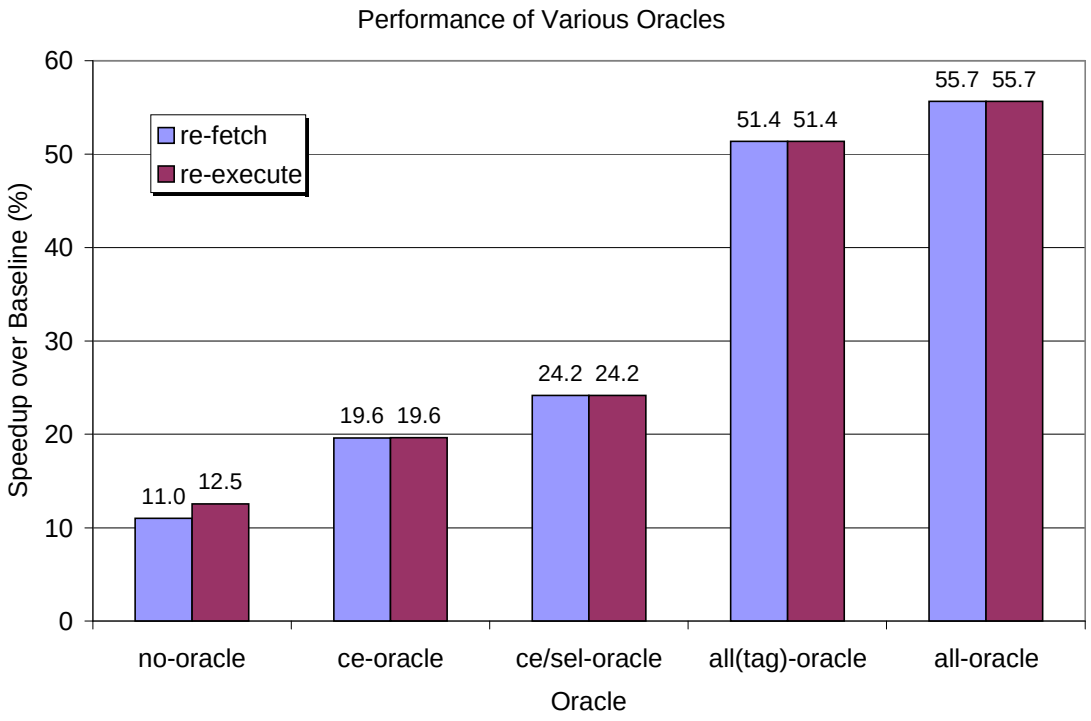

Figure 7.4: Performance of L4V predictors with different oracles.

Adding perfect confidence estimation (*ce-oracle*) results in a significant increase in speedup in comparison with the implementable predictor (*no-oracle*). Because the *no-oracle*'s accuracy is already very high (mid to high

nineties), the performance improvement most likely does not stem from inhibiting the few incorrect predictions that the *no-oracle* predictor makes but rather from the relatively large number of times the *no-oracle* does not make a prediction even though it has the correct information available. Since the CE setting of the *no-oracle* predictor is the result of a global optimization and therefore yields the highest speedup, I conclude that trading off missing potentially correct predictions for reducing the number of incorrect predictions is beneficial in the simulated CPU. Apparently, incorrect predictions incur a high cycle penalty and should therefore be avoided, which results in a conservative predictor with a high accuracy but a relatively low coverage. Note that, because there are no mispredictions and hence no recoveries, the *ce-oracle* speedups for re-fetch and re-execute are (almost) the same. They are not exactly the same because a quite different *SAg* CE setting is used for re-fetch than for re-execute, which affects the performance of the (imperfect) selector. The more precise speedup numbers are 19.614% for re-fetch and 19.643% for re-execute, showing that rather different CE settings result in similar but not quite identical selector performance.

Perfect confidence estimation in combination with perfect selection (*ce/sel-oracle*) boosts the speedup even more. Clearly, the selection mechanism used in the *no-oracle* predictor is not perfect. Overall, the L4V predictor's confidence estimator and selector are able to reap 45% to 52% of the theoretically possible speedup for this predictor (*no-oracle* versus *ce/sel-oracle* speedup).

A comparison with the perfect load value predictor (*all-oracle*), however, shows that there is still a large amount of potential for improvement left. The predictor only yields 20% to 23% of the speedup that can theoretically be attained with load value prediction. Comparing the *all-oracle* with the *ce/sel-oracle* shows that the L4V predictor does not even contain the necessary information to reach half the possible speedup. This large gap suggests that there exists significant opportunity for other prediction methods. It is, how-

ever, unclear how much of the remaining potential can be realized, in particular with a limited amount of state for storing information.

Comparing *all(tag)-oracle* with *all-oracle* shows that having only 512 predictor lines does not hamper the performance much due to aliasing, which was to be expected since the last two value predictor with 1024 lines performs worse than the L4V predictor of the same size (see Section 7.3). The average loss of prediction potential over SPECint95 due to tag misses is less than eight percent in a 512-line load value predictor.

7.5 *SAg* L4V Sensitivity Analysis

So far, the different load value predictors have been optimized to yield the highest harmonic mean speedup over the eight benchmark programs. In this section, the L4V predictor's performance will be optimized for each individual program separately. Furthermore, the sensitivity of the *SAg* history length and counter size is analyzed.

Unless otherwise noted, the load value predictor used is an eight-bit partially tagged, *SAg*-based last four value predictor with a capacity of 2048 values (requiring sixteen kilobytes of state for retaining load values). The *SAg* counter top is sixteen for re-fetch and eight for re-execute. The threshold and penalty values are optimized and differ for each case as indicated.

7.5.1 *SAg* History Length

I have already shown the *SAg* confidence estimator to work well with ten-bit histories in the last value predictor [BuZo98b]. Figure 7.5 shows the performance of the last four value predictor for different history lengths. Note that, in order to make the trend more apparent, the figure is not zero-based.

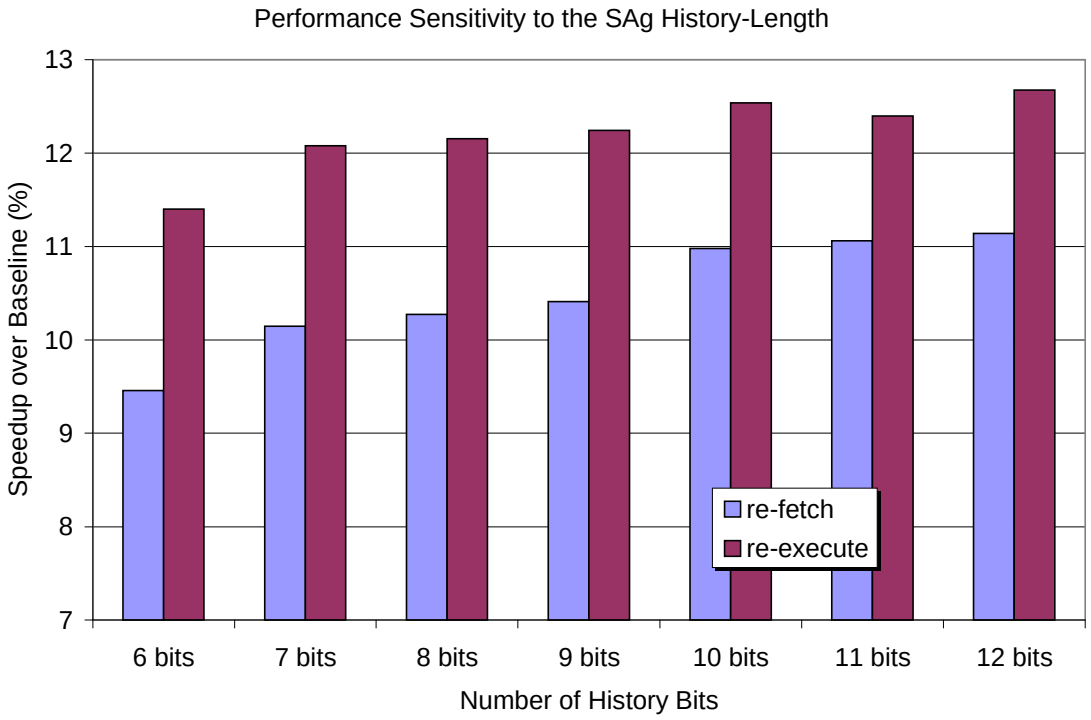

Figure 7.5: Mean speedup with different history lengths.

As the figure illustrates, ten-bit histories also mark the beginning of the performance saturation for the last four value predictor, both with re-fetch and re-execute. Similar investigations of several other predictors with *SAg* confidence estimators show that ten-bit histories are sufficient for most *SAg*-based load value predictors.

Note that it is important to chose as short a history as possible because every additional bit doubles the number of required saturating counters in the second level of the *SAg* confidence estimator. Hence, to maximize the performance while keeping the number of history bits at a minimum, I chose ten-bit histories for basically all of my predictors.

Note that all the predictors in Figure 7.5 use the same threshold and penalty values. Only the number of history bits and therefore the number of saturating counters is varied. A threshold of fourteen and a penalty of eleven for re-fetch and a threshold of seven and a penalty of four for re-execute represents the optimum for the ten-bit case. Using these parameters with the longer and shorter histories does not necessarily result in the best performance. However, the results from Section 7.5.3 indicate that the performance is most likely very close to optimal. Nevertheless, this suboptimality is probably the reason why the eleven-bit re-execute performance is slightly lower than its ten-bit counterpart. Note that while the expected optimal performance in the nine-bit case is slightly higher than shown, it is not as high as the performance in the ten-bit case.

7.5.2 *SAg* Counter Parameters

Figure 7.6 illustrates how well the last four value predictor performs with differently sized saturating counters in the *SAg* CE. Note that the presented performances are obtained with an optimized threshold and penalty value for each predictor configuration. Again, the trends seen in the figure are representative of other *SAg*-based load value predictors as well.

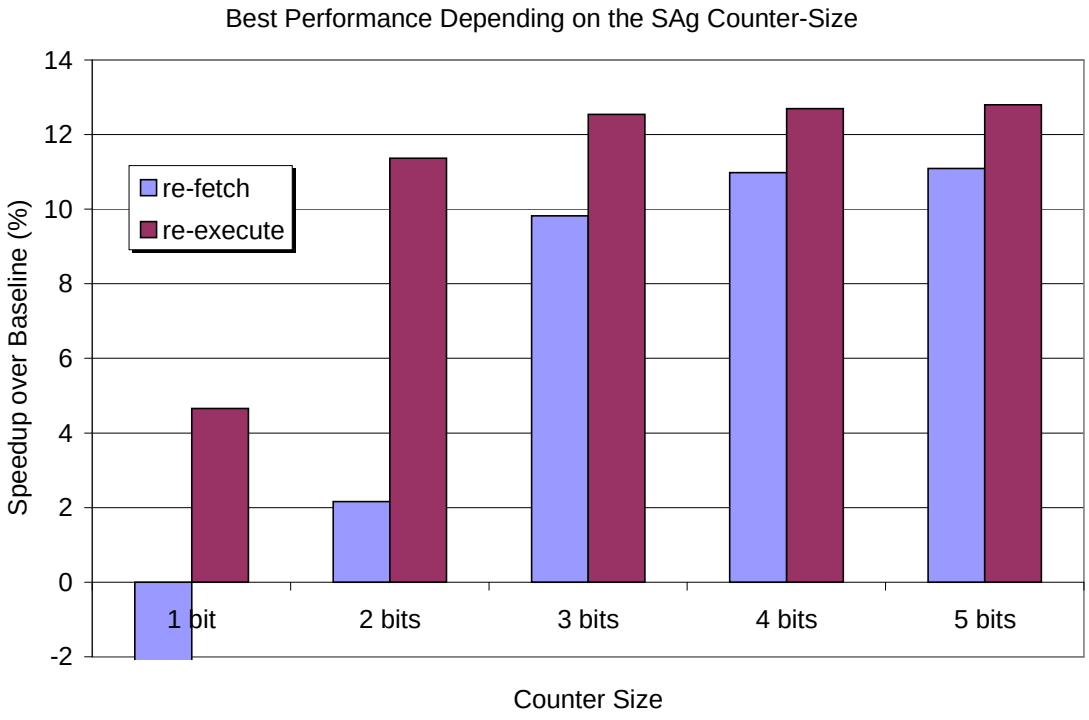

Figure 7.6: Best L4V performance for different saturating-counter sizes.

As the figure shows, counter top values smaller than eight (three bit counters) for re-execute and sixteen (four bit counters) for re-fetch diminish the achievable performance, while larger counters do not significantly improve the performance. To keep the counters as small as possible without affecting the performance overly much, I selected three-bit counters for re-execute and four-bit counter for re-fetch for all predictors in this dissertation (unless otherwise indicated).

7.5.3 Optimizing Individual Programs

So far, all the presented performance numbers have been harmonic mean speedups over the eight SPECint95 benchmark programs. While these numbers are hopefully representative of the average benefit one can expect from adding a load value predictor to a CPU, the actual performance improvements of individual programs do vary substantially.

Figure 7.7 and Figure 7.8 show the individual speedups of the eight benchmark programs for re-fetch and re-execute, respectively, as well as the harmonic mean speedup over the entire suite. Two results are given for each program. The left bar shows the speedup of each program using the last four value predictor configuration that yields the highest average speedup, whereas the right bar shows the highest individual speedup, i.e., when the predictor's threshold and penalty values are optimized for each program separately.

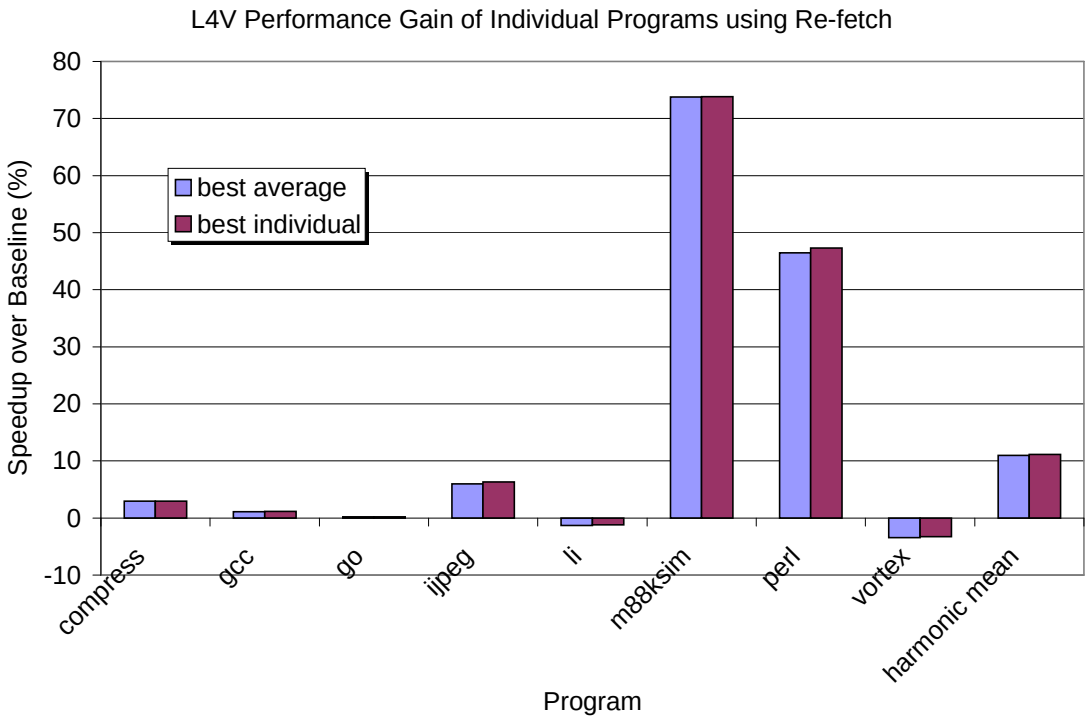

Figure 7.7: The speedup of the SPECint95 programs using re-fetch.

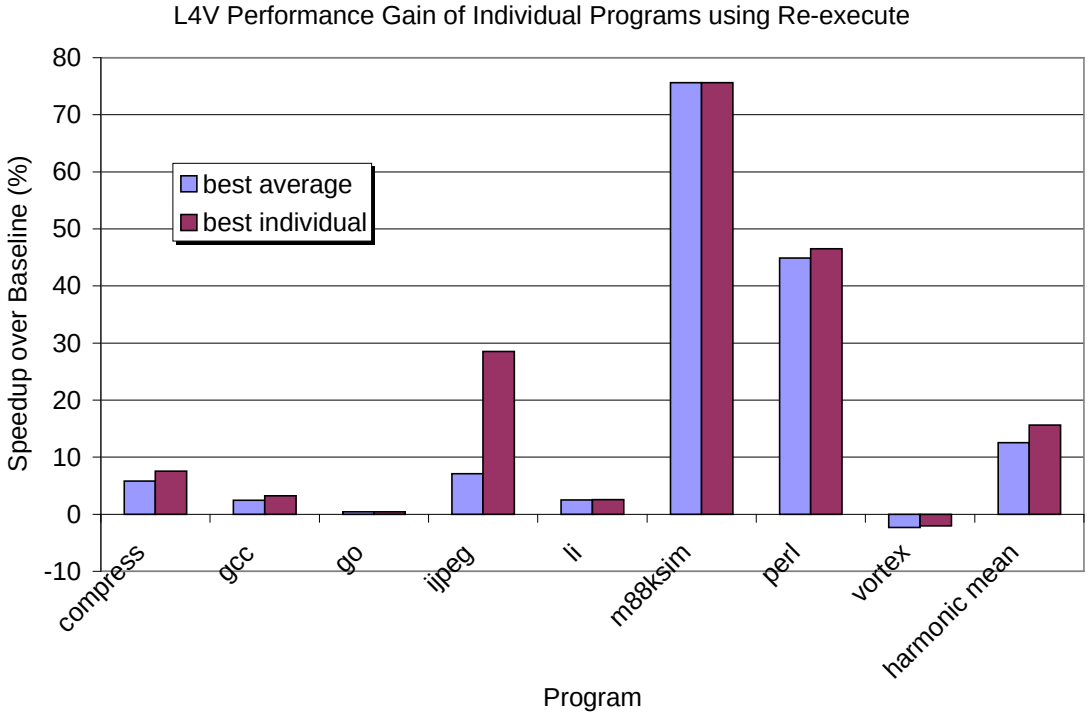

Figure 7.8: The speedup of the SPECint95 programs with re-execute.

Clearly, *perl* and *m88ksim* benefit the most from load value prediction. The speedup of the remaining programs is moderate. Surprisingly, no configuration of the *SAg*-based last four value predictor yields a positive speedup for *vortex*. Since *vortex* is a database program, I can only assume that it simply does not exhibit sufficiently short and discernable predictability patterns that the predictor could exploit. With re-fetch, the program *li*, which is a Lisp interpreter, can also not be sped up by the L4V load value predictor.

It is quite evident in both figures that the speedups of the individual programs vary significantly. What is almost more surprising, though, is that the individually optimized predictor configurations yield only marginally more performance than the best average speedup with a re-fetch misprediction recovery mechanism. This is a promising result because it shows that a single predictor configuration fits most programs reasonably well.

Generally speaking, the same is also true with re-execute recovery. However, the re-execute speedups differ more between the individually optimized programs and the best average case than the re-fetch speedups. The reason for the large discrepancy between the individual and average harmonic mean speedups for re-execute is the program *ijpeg*. For some reason, this program benefits very much from a better threshold and penalty setting. Clearly, the setting that yields the best average performance does not work well for *ijpeg*.

I conclude that adapting the confidence estimator setting to individual programs is only seldom necessary and that a good predictor configuration generally yields good results for most programs.

Of course one would expect all the programs to perform better with re-execute than with re-fetch. This is indeed so for seven of the eight programs. However, *perl* performs slightly better with re-fetch than with re-execute. The reason is that for *perl* the re-execute counter top of eight is too restrictive, which is why it performs better with the re-fetch counter top of sixteen in spite of the more costly recovery mechanism. This result shows that the selected

counter tops of eight and sixteen are not necessarily good for all programs and that performance analyses are required to obtain the ideal counter top value for individual programs.

Table 7.1 lists the fixed counter top values and the best threshold and penalty value of the *SAg* L4V predictor for each SPECint95 program as well as the parameters that result in the best average performance.

		compress	gcc	go	ijpeg	li	m88ksim	perl	vortex	best avg
re-fetch	counter top	16	16	16	16	16	16	16	16	16
	threshold	11	15	15	12	15	14	13	15	14
	penalty	4	6	9	4	13	12	3	13	11
re-exec	counter top	8	8	8	8	8	8	8	8	8
	threshold	0	6	7	0	7	7	6	7	7
	penalty	n/a	1	4	n/a	7	4	1	7	4

Table 7.1: Best individual and average predictor configurations.

With re-fetch, we find that all the programs have a threshold close to the best average threshold of fourteen. With re-execute, *compress* and *ijpeg* are outliners in the sense that they both perform best with a threshold of zero, which disables the confidence estimator altogether (but not the selector) and allows every load to be predicted. In other words, these two programs perform best without a confidence estimator. Note that while a penalty value is not applicable (n/a) with a threshold of zero as far as the confidence estimator is concerned, the chosen penalty value of one still affects the performance of the selector. However, the impact of different penalties on the selector performance is only minimal, as was noted in Section 7.4.1.

Unlike the threshold values, the optimal penalty values vary considerably from one program to another. However, there is a clear correlation between the best re-fetch and re-execute penalties because whenever the best penalty value is low for one recovery mechanism then it is also low for the other recovery mechanism and vice-versa (relative to the counter top value).

While the penalties listed in Table 7.1 represent the best values for the given predictor and workload, changing them quite drastically does not impact

the performance much, as the results from Figure 7.7 and Figure 7.8 indicate. Table 7.2 illustrates this weak dependence between the speedup and the selected penalty value for the program *gcc*. It shows the speedup of *gcc* for different penalty values when all the other parameters are held constant.

	Re-fetch speedup of gcc using a Tag SAg L4V predictor with 512 lines, a counter top of 16 and a threshold of 15														
penalty	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
speedup	-1.44	0.67	1.06	1.15	1.17	1.18	1.16	1.16	1.14	1.13	1.13	1.10	1.10	1.09	1.08

Table 7.2: The L4V speedup of *gcc* for different penalty values.

The highest speedup is obtained with a penalty of six. For both higher and lower penalties the speedup drops slightly. However, any penalty between three and fifteen results in close to optimal performance. Only penalties of one and two yield significantly inferior performance. This behavior of a stable performance over a wide range of penalties (sometimes with drop-offs at either end) is quite typical for most load value predictors, as the speedup maps in Appendix C verify (see Section 8.2 for the definition of a speedup map).

Table 7.3 shows that the dependence between the speedup and the threshold value is more pronounced than the dependence between the speedup and the selected penalty.

	Re-fetch speedup of gcc using a Tag SAg L4V predictor with 512 lines, a counter top of 16 and a penalty of 11														
threshold	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
speedup	-23.30	-10.77	-4.56	-1.84	-0.50	0.15	0.53	0.75	0.89	0.98	1.04	1.07	1.10	1.10	1.13

Table 7.3: The L4V speedup of *gcc* for different threshold values.

With a penalty of eleven, the best threshold is fifteen. Threshold values above about eight yield a decent performance, but lower thresholds significantly affect the predictor's effectiveness. In fact, the two lowest thresholds result in a substantial slowdown, illustrating the need for a good threshold value. Nevertheless, the performance does not abruptly drop off near the op-

timal threshold value. Rather, thresholds near the optimal value still result in relatively good performance. Again, this behavior is rather typical for load value predictors, as the speedup maps in Appendix C show.

7.5.4 Using Distinct Last Values

Instead of simply retaining the last four values, Wang and Franklin propose a last *distinct* four value predictor [WaFr97], which also stores four values per line, but the predictor only inserts a new value if that value is not already among the four values. Unfortunately, finding out whether the value is already present requires content addressable memory, which storing the last four values regardless of whether any of them are identical does not.

In previous work [BuZo99b] and Section 9.4.1, I show that the regular last four value predictor with its lower complexity outperforms Wang and Franklin's last distinct four value predictor in spite of the latter's somewhat higher predictability potential. Figure 7.9 shows the last *n* value predictability when retaining every loaded value versus only retaining distinct values. The potential is given as the percentage of the fetched load values that are identical to at least one of the retained values.

According to Figure 7.9, larger *n* yield a higher predictability potential. This result is intuitive since the chance of finding the correct value increases as the number of retained values becomes larger. The increase is considerable for small *n* up to about four. Then the "curve" starts flattening out and saturates at approximately $n = 11$, at which point almost no extra potential is gained by further increasing *n*.

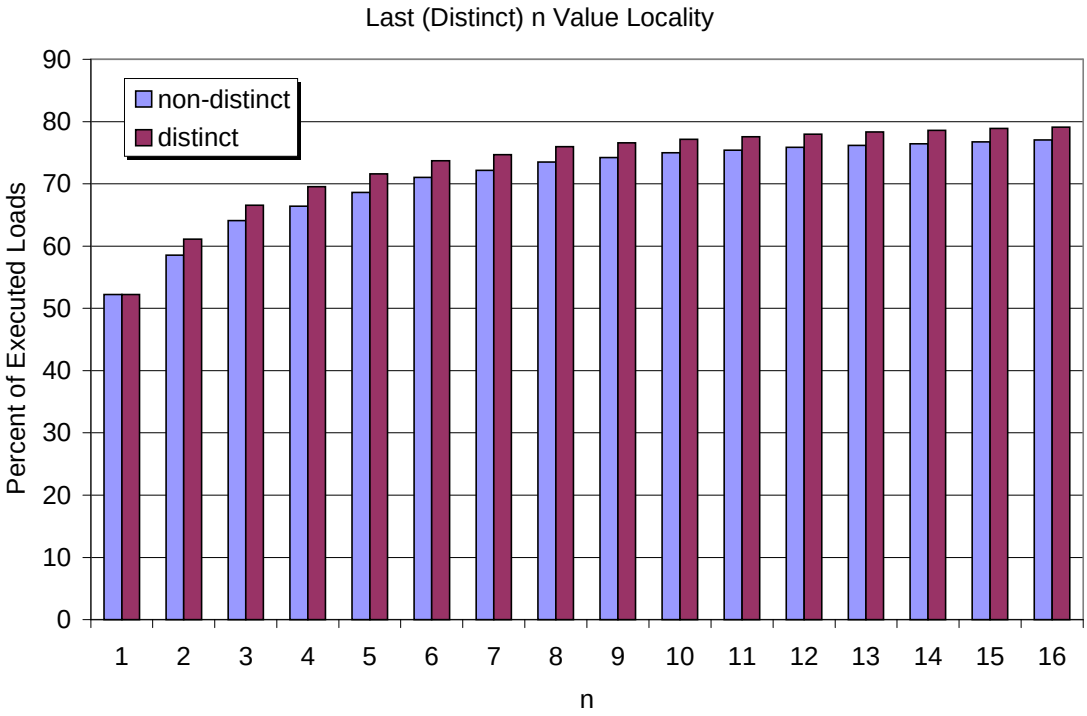

Figure 7.9: The average last *n* value and last distinct *n* value predictability.

One interesting observation is that the potential difference between distinct and non-distinct values is virtually constant. Hence, the relative advantage of storing distinct values becomes smaller as *n* gets larger.

For *n* = 4, which is the predictor width Wang and Franklin chose [WaFr97], the potential difference is one eighteenth of the total potential of about sixty-four percent. Therefore, a predictor that retains the last four not necessarily distinct values is theoretically able to perform nearly as well as its more complex counterpart that retains the last distinct four values.

Wang and Franklin's predictor uses a least recently used replacement policy and a *bimodal* CE (and selector) that is indexed by a usage-pattern. Their predictor only outperforms the Tag *SAg* L4V predictor for very small predictor sizes (see Section 9.4.1). To see whether it is possible to reap the additional potential that lies in retaining only distinct values in larger predictors as well, I modified the Tag *SAg* L4V predictor so that it only inserts a new value if the new value is not already among the four stored values. In all other aspects the new predictor (called Ld4V) operates just like the old one (L4V), i.e., it has a *SAg* CE in each of the four sub-components that are also used as the selector. The performance of this new predictor is presented in Figure 7.10 for three predictor sizes (in number of total values retained).

Retaining only distinct values is clearly beneficial. With re-execute recovery, larger predictors appear to benefit less from retaining only distinct values. The opposite seems to be true for re-fetch. The reason why Wang and Franklin's predictor with its least recently used replacement policy (the Ld4V uses first-in first-out) does not perform as well as the comparable L4V and Ld4V predictors is probably the usage-pattern-based *bimodal* confidence estimator. The performance of their predictor is shown in Section 9.4.1.

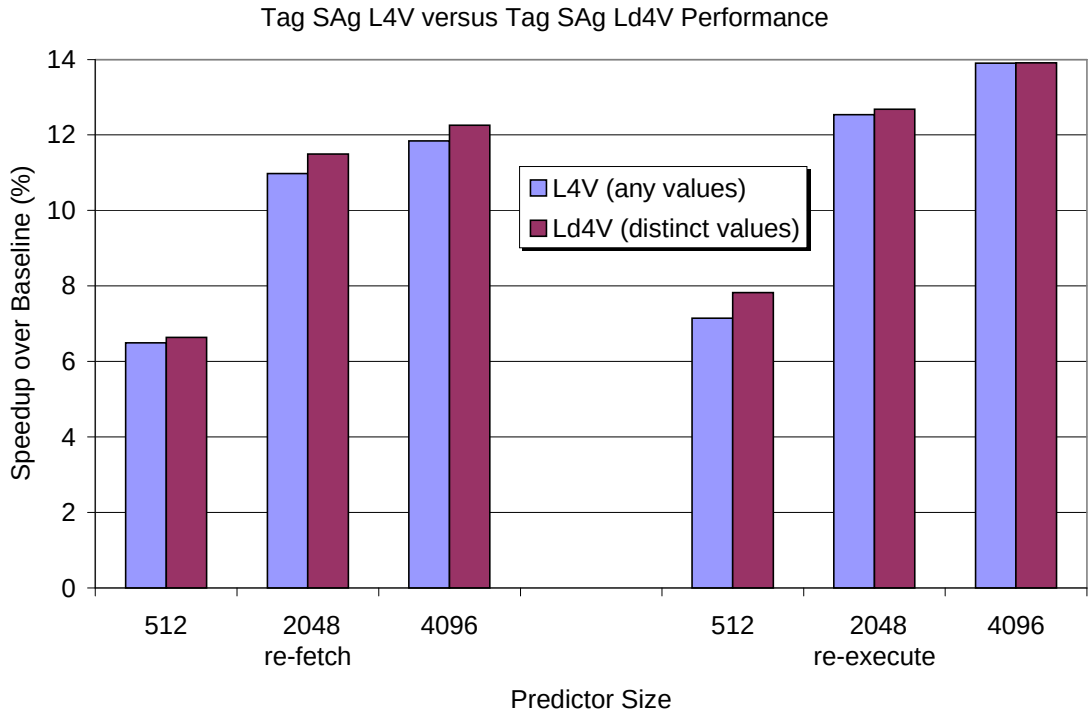

Figure 7.10: Speedup of the Tag *SAg* L4V and the Tag *SAg* Ld4V.

7.6 Summary

This chapter shows that the highly skewed distribution of the execution frequency of load instructions results in an unbalanced utilization of the load value predictor hardware. As a result, it can be advantageous to make load value predictors wider and shorter, that is, to retain more information about few load instructions instead of retaining little information about a large number of loads. For example, among last *n* value predictors with a capacity of 2048 load values, the last four value predictor represents the best width versus height trade-off for SPECint95 and yields the highest speedup. The larger the predictor, the larger a predictor width results in the best performance.

In addition, this chapter also investigates the sensitivity of the L4V predictor's performance of various predictor parameters. For instance, three to four-bit saturating counters and ten-bit histories seem to be necessary for good *SAg* CE performance. Larger counters and histories do not result in significantly higher speedup for most programs. Furthermore, the performance delivered by a load value predictor varies greatly from program to program. Nonetheless, the predictor configuration that yields the best average performance over the SPECint95 benchmark suite also yields close to optimal performance for each of the individual programs with only one significant exception. Finally, retaining only distinct values in a last four value predictor instead of retaining values regardless of whether some of them are identical appears to be somewhat beneficial, in particular with small predictor sizes.

Chapter 8

Hybridizing Load Value Predictors

This chapter investigates the benefit of building predictors that combine several load value predictors in one, i.e., of hybridizing predictors. The next chapter will show how to reduce the size of hybrid load value predictors.

8.1 The Benefit of Hybridization

It is not a priori clear whether combining multiple load value predictors results in a predictor that is capable of predicting more load instructions correctly or that can make more accurate predictions. For example, it may happen that two different predictors essentially predict the same load instructions with the same values. Obviously, combining two such predictors would not result in improved performance but only in a larger and more complex predictor. As mentioned in Section 4.3.3, the stride (2-delta) predictor is able to make last value predictions with a stride value of zero. Consequently, combining a last value predictor with a stride predictor will probably not yield a predictor that is more effective than the stride predictor by itself.

Like the last *n* value predictor from the previous chapter, hybrid predictors consist of multiple component predictors of which one component must be selected for making a prediction. The confidence estimators are used to guide the selection process, i.e., the component that reports the highest confidence is selected but the selected component is only allowed to make a prediction if its confidence is above the preset threshold.

8.2 Hybrid Performance

In order to determine which predictors complement each other well in a hybrid configuration, I tested every possible combination between a register, last value, stride 2-delta, last four value, and finite context method predictor. Because the last four value predictor is a strict superset of the last value predictor (with the same number of predictor lines), hybrid combinations that include both a LV and a L4V predictor are excluded from this study. The performance of the excluded hybrids is exactly the same as the performance of the same predictor without the LV component.

The components in the hybrid predictors discussed in this chapter are prioritized to resolve selector ties, i.e., when two or more components report the same highest confidence. In such a case, the component with the highest confidence and the highest priority is selected. If only one component reports the highest confidence, then that component is selected regardless of its priority. Since changing the priority among the components of a hybrid does not appear to affect the performance considerably (see Section 9.5.1), only hybrids in which the components are prioritized in the following order (from high priority to low priority) are investigated: Reg, LV, St2d, L4V, FCM.

Because it is the goal of this chapter to study which predictors complement each other well, all components are 2048 lines tall regardless of the resulting hybrid's size. This height was chosen because such predictors already yield a performance that is close to the performance of the same predictor with an infinite number of lines. (This observation is also supported by the quantile numbers from Figure 7.1.) Hence, studying hybrids of 2048-line components should suffice to identify the most promising combinations for building high-performing hybrid load value predictors.

While the size of some of the resulting hybrids is rather large, they can frequently be made smaller by sharing state between their components (see Chapter 9). Nevertheless, due to the varying predictor sizes, care must be taken when using the performance numbers shown in this section for inter-

hybrid comparisons.

Figure 8.1 shows the performance of all hybrid combinations with *bimodal* and *SAg* confidence estimators (and selectors) when a re-fetch misprediction recovery mechanism is used. The predictors are sorted by increasing *SAg* performance. The hybrid's names are character combinations in which each character represents one component: *r* stands for register, *l* for last value, *s* for stride 2-delta, *4* for last four value, and *f* for finite context method predictor.

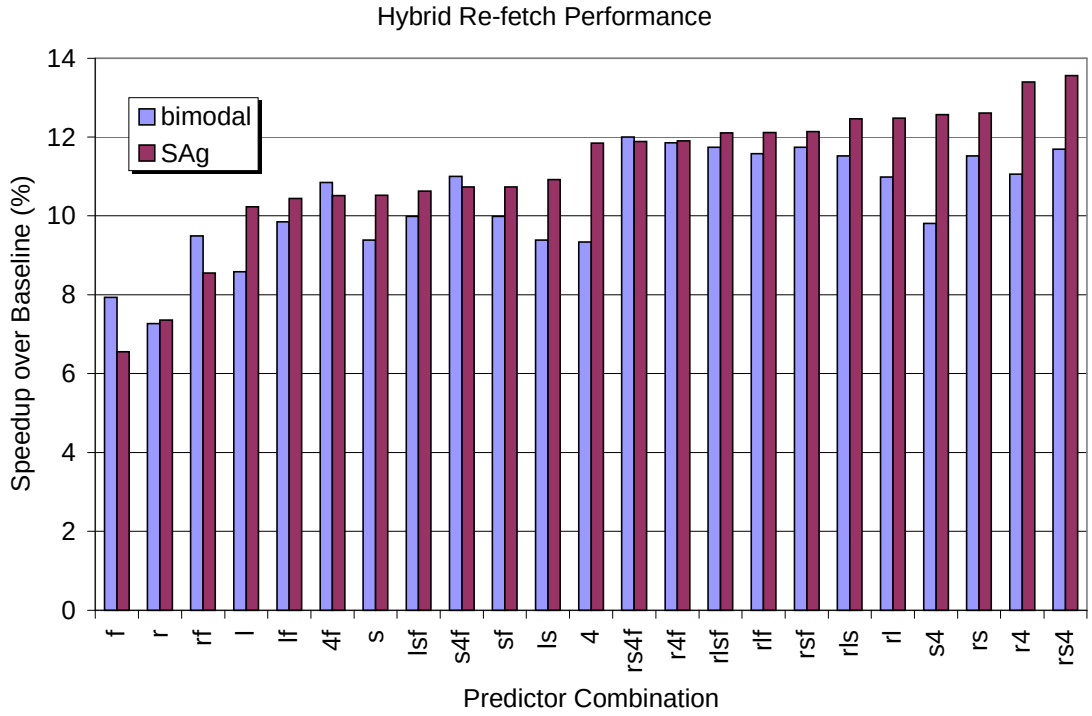

Figure 8.1: Hybrid performance using re-fetch.

Note that it is not feasible to optimize the threshold and penalty for every hybrid individually. Instead, the threshold and penalty values that yield the highest average speedup over the included components are used as an approximation. This average speedup is computed as follows. I evaluated the speedup of the five basic predictors for a large number of threshold and pen-

alty pairs and recorded the results in speedup maps, which are shown in Appendix C. A speedup map is essentially a matrix with different thresholds in one dimension and different penalties in the other dimension. The matrix elements are the speedups measured for the threshold and penalty that intersect at that element. Then an average map is computed by forming the arithmetic mean of the entries in the individual maps of each included component (e.g., the register predictor's map and the last value predictor's map for the *rl*-hybrid). The highest speedup in the averaged map determined the threshold and penalty value used for the given hybrids. Note that this approach does not always yield the best performing hybrid but is usually close. For example, the St2d+FCM hybrid yields a speedup of 9.99% with re-fetch and 13.09% with re-execute when using the parameters from the averaged speedup map, whereas truly optimizing the threshold and penalty values results in a speedup of 10.01% for re-fetch and 13.94% for re-execute.

The confidence estimator configurations derived from the averaged speedup maps are summarized in Table 8.1. The counter top value is sixteen for re-fetch and eight for re-execute.

			4	4f	f	l	lf	ls	lsf	r	r4	r4f	rf	rl	rlf	rls	rlsf	rs	rs4	rs4f	rsf	s	s4	s4f	sf
SAg	re-fetch	threshold	15	15	15	13	15	12	14	15	15	15	15	13	15	14	15	11	15	15	15	12	15	15	15
		penalty	8	11	11	5	9	5	6	7	8	10	10	8	9	5	7	7	7	7	7	5	7	7	7
	re-exec	threshold	7	7	6	5	6	5	5	4	7	6	6	6	5	5	5	5	6	5	5	5	6	6	5
		penalty	3	3	3	2	3	1	2	1	3	3	3	1	2	2	2	1	1	2	2	1	1	2	2
bimodal	re-fetch	threshold	15	13	13	10	13	11	13	10	13	13	13	10	13	11	13	11	12	13	13	12	12	13	13
		penalty	13	11	11	15	11	12	11	7	12	11	11	7	11	12	11	12	12	11	11	12	12	11	11
	re-exec	threshold	6	7	7	5	4	5	5	2	6	7	4	4	4	5	5	5	5	4	5	5	5	6	3
		penalty	3	3	3	1	3	1	2	2	3	3	3	2	3	1	2	1	2	3	2	1	2	3	4

Table 8.1: The confidence estimator parameters of the hybrid predictors.

As has been determined in Section 5.4, with re-fetch the FCM predictor performs better with a *bimodal* than a *SAg* confidence estimator. Because of that, some of the *bimodal* hybrids that contain an FCM component outperform their *SAg*-based counterparts, as Figure 8.1 illustrates. However, most hybrids benefit from having a *SAg* CE and all the high speedups are obtained with *SAg* CEs.

Hybrids with more components tend to yield more speedup than the ones with fewer components. This is particularly true for the *bimodal* hybrids. However, there are many notable exceptions with the *SAg*-based hybrids. For instance, the speedup of the best performing predictor (Reg+St2d+L4V) decreases when adding an FCM component to it. Likewise, Reg+L4V, Reg+St2d, St2d+L4V, Reg+LV, Reg+LV+St2d, L4V, and LV+St2d all suffer when an FCM component is included. Only the Reg, the LV, and the St2d predictors benefit from an FCM. This is clearly a result of the poor performance of the *SAg* CE in connection with the FCM predictor. Maybe hybridizing with a *bimodal* FCM would result in a better performance. However, the investigation of heterogeneous hybrids in which the components use different kinds of confidence estimators is left for future work.

Aside from the FCM component, there also exist other “anomalies”. For example, the Reg+St2d predictor outperforms the Reg+LV+St2d predictor. The Reg+L4V+FCM, the Reg+LV+FCM, and the Reg+LV predictors do not benefit from having an St2d component added to them. Furthermore, the speedup of the Reg+St2d+FCM and the St2d+FCM predictors decreases when adding a L4V component to them. The reason for this counterintuitive behavior is negative interference.

Because adding a component to a hybrid makes the task of the selector harder (there are more choices), it can happen that the additional predictability potential is unable to offset the increased selector-related losses. When that situation occurs, the hybrid’s components interfere negatively with one another and lower the overall performance.

Note that some of the most effective hybrids are small hybrids with only two components (Reg+LV and Reg+St2d). The remaining three of the five best combinations are significantly larger because they include an L4V component. However, Chapter 9 demonstrates how the size of a Reg+St+L4V predictor can be reduced to only slightly more than that of a Reg+St2d hybrid basically without loss of performance.

Note also that eleven of the twelve best performing *SAg* hybrids include the storage-less register predictor, indicating that the Reg predictor is a very important component in a hybrid. This result is particularly surprising because the Reg predictor by itself performs only poorly. Note that no profiling was used to adapt the register allocation, which can significantly improve the performance of the Reg predictor [TuSe99], yet the benefit from including a register value predictor is already substantial.

Figure 8.2 shows the performance of all hybrid combinations with a re-execute misprediction recovery mechanism. Again, the predictors are sorted by increasing *SAg* performance.

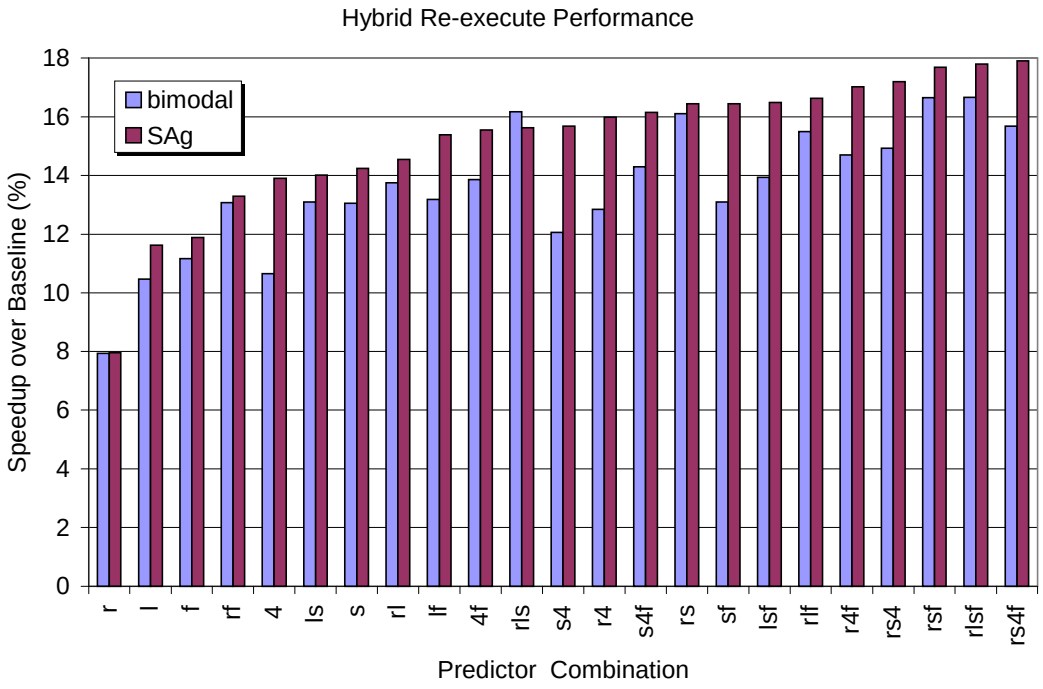

Figure 8.2: Hybrid performance using re-execute.

Clearly, hybrid load value predictors have the potential to yield significantly higher speedups than even the best single-component predictors.

With re-execute, the Reg+LV+St2d hybrid is the only *bimodal* predictor that outperforms its *SAg* counterpart. It is not clear why that is and it could be

an artifact of not truly optimizing the predictors' CE configurations. Nevertheless, all the other hybrids perform better with a *SAg* CE than a *bimodal* CE and the highest speedups are again obtained with a *SAg* CE.

There is also negative interference with a re-execute misprediction recovery mechanism. Again, the L4V component diminishes the performance when it is included in the St2d+FCM predictor, and the Reg+St2d predictor suffers when an LV component is added to it. There is one new case of negative interference that was not present with re-fetch. The St2d predictor outperforms the LV+St2d hybrid with re-execute.

Clearly, the problem with the FCM component is gone because with re-execute even the FCM predictor performs better with a *SAg* CE than with a *bimodal* CE (see Section 5.4). In fact, seven of the eight best performing hybrids include an FCM, six of them include an St2d, and six include a Reg component. The best performing re-fetch hybrid (Reg+St2d+L4V) is among the four best performing re-execute hybrids and is the only one of them that does not include an FCM component, which may be significant because the FCM is a two-level value predictor whereas the other four studied basic predictors comprise only one level.

With re-execute, there is a tendency showing that the more components a predictor has the more it benefits from a *SAg* CE as opposed to a *bimodal* CE. This may indicate that the *SAg* CE not only makes better confidence estimations but also better selections than the *bimodal* confidence estimator.

When averaging the re-fetch and the re-execute speedups of the *SAg* hybrids, the Reg+St2d+L4V predictor turns out to perform best by a considerable margin. The most effective two-component hybrid is the Reg+L4V, which is closely followed by the Reg+St2d hybrid. The best single-component predictor is L4V, which is trailed by the St2d predictor. None of the four-component hybrids outperform the best three-component hybrid.

Because of its generally inferior performance and to reduce the number of data-points, I will disregard the *bimodal* CE for the rest of this chapter.

Table 8.2 and Table 8.3 show the results from Figure 8.1 and Figure 8.2, respectively, in a different form. Both tables list the hybrids and their speed-ups over the baseline processor on the left. The numbers on the right represent the increase in speedup percentage points when adding the given components to the listed hybrids.

Clearly, both with re-fetch and re-execute, all the hybrids that do not include Reg would benefit considerably from having a Reg component. This is particularly surprising because the Reg predictor does not perform very well when used in isolation. Similarly, the Reg predictor benefits from being augmented with any other component. Except for the Reg predictor, only the FCM and Reg+FCM hybrids benefit considerably from an LV component. These two predictors also profit the most from having a St2d or a L4V component added to them. As mentioned earlier, most hybrids are slowed down by an FCM component with re-fetch, whereas it is advantageous for most hybrids to have an FCM component with re-execute. Several predictors benefit from an L4V component, which is understandable because the L4V and the FCM are the largest single-component predictors.

Re-fetch Speedup Benefit of Adding Components to SAg Hybrids						
hybrid	speedup	+r	+l	+s	+4	+f
r	7.4		5.1	5.3	6.0	1.2
l	10.2	2.2		0.7	1.6	0.2
rl	12.5			0.0	0.9	-0.4
s	10.5	2.1	0.4		2.0	0.2
rs	12.6		-0.1		0.9	-0.5
ls	10.9	1.5			1.6	1.2
rls	12.5				1.1	-0.4
4	11.8	1.5	0.0	0.7		-1.3
r4	13.4		0.0	0.2		-1.5
s4	12.6	1.0	0.0			-1.8
rs4	13.6		0.0			-1.7
f	6.6	2.0	3.9	4.2	4.0	
rf	8.6		3.6	3.6	3.4	
lf	10.4	1.7		0.2	0.1	
rlf	12.1			0.0	-0.2	
sf	10.7	1.4	-0.1		0.0	
rsf	12.1		0.0		-0.3	
lsf	10.6	1.5			0.1	
rlsf	12.1				-0.2	
4f	10.5	1.4	0.0	0.2		
r4f	11.9		0.0	0.0		
s4f	10.7	1.2	0.0			
rs4f	11.9		0.0			

Table 8.2: Re-fetch speedup benefit from adding components.

Re-execute Speedup Benefit of Adding Components to SAg Hybrids						
hybrid	speedup	+r	+l	+s	+4	+f
r	8.0		6.6	8.5	8.0	5.3
l	11.6	2.9		2.4	2.3	3.8
rl	14.5			1.1	1.4	2.1
s	14.2	2.2	-0.2		1.5	2.2
rs	16.4		-0.8		0.7	1.2
ls	14.0	1.6			1.7	3.7
rls	15.6				1.6	2.2
4	13.9	2.1	0.0	1.8		1.7
r4	16.0		0.0	1.2		1.0
s4	15.7	1.5	0.0			0.5
rs4	17.2		0.0			0.7
f	11.9	1.4	3.5	4.6	3.7	
rf	13.3		3.3	4.4	3.7	
lf	15.4	1.2		1.1	0.2	
rlf	16.6			1.2	0.4	
sf	16.4	1.2	0.0		-0.3	
rsf	17.7		0.1		0.2	
lsf	16.5	1.3			-0.3	
rlsf	17.8				0.1	
4f	15.6	1.5	0.0	0.6		
r4f	17.0		0.0	0.9		
s4f	16.2	1.8	0.0			
rs4f	17.9		0.0			

Table 8.3: Re-execute speedup benefit from adding components.

8.3 Shared and Unique Performance Contributions

In an effort to determine why the Reg predictor is such a valuable addition to all hybrids while the LV component, for example, generally is not, I investigated how frequently each component in a hybrid is able to make a correct prediction that none of the other components can make, how often the predictions from the individual components overlap, and how often they interfere with one another. Because not every prediction is equally important (e.g., predicting a load that hits in the L1 data-cache is not as important as predicting a load that has to go all the way to memory), the following subsections study the speedup contributions of the individual hybrid components rather than the actual set of load instructions that each component is able to predict.

8.3.1 Two-component Hybrids

A hybrid component's unique speedup contribution is the part of the overall performance that is lost when that component is removed. Hence, the component must actually be present to deliver its unique performance contribution. Conversely, in a two-component hybrid, the shared contribution is common to both components, meaning that either one is able to provide the contribution, but the contribution does no increase if both components are used together. Therefore, only one of the two components is needed to deliver the shared performance contribution.

The unique and shared speedup contributions in two-component hybrids are computed as follows. Assuming that predictor *A* yields a speedup of *a* when used in isolation, predictor *B* yields a speedup of *b*, and the hybrid predictor *A+B* yields a speedup of *c*, then *A* contributes *c-b* unique speedup, *B* contributes *c-a* unique speedup, and the shared contribution is *a+b-c*.

Venn-diagrams are used to visualize the different contributions. For example, the top left Venn-diagram in Figure 8.3 indicates that with re-execute

recovery, twenty percent of the *SAg*-based Reg+LV hybrid's speedup stems uniquely from the Reg component, 45.2% from the LV component, and the shared contribution is about 34.7%. (The sum of the three contributions is one-hundred percent, but sometimes the rounding to one digit after the decimal point makes it appear otherwise.) Figure 8.3 shows results for all two-component hybrids (the L4V predictor is treated as a single component).

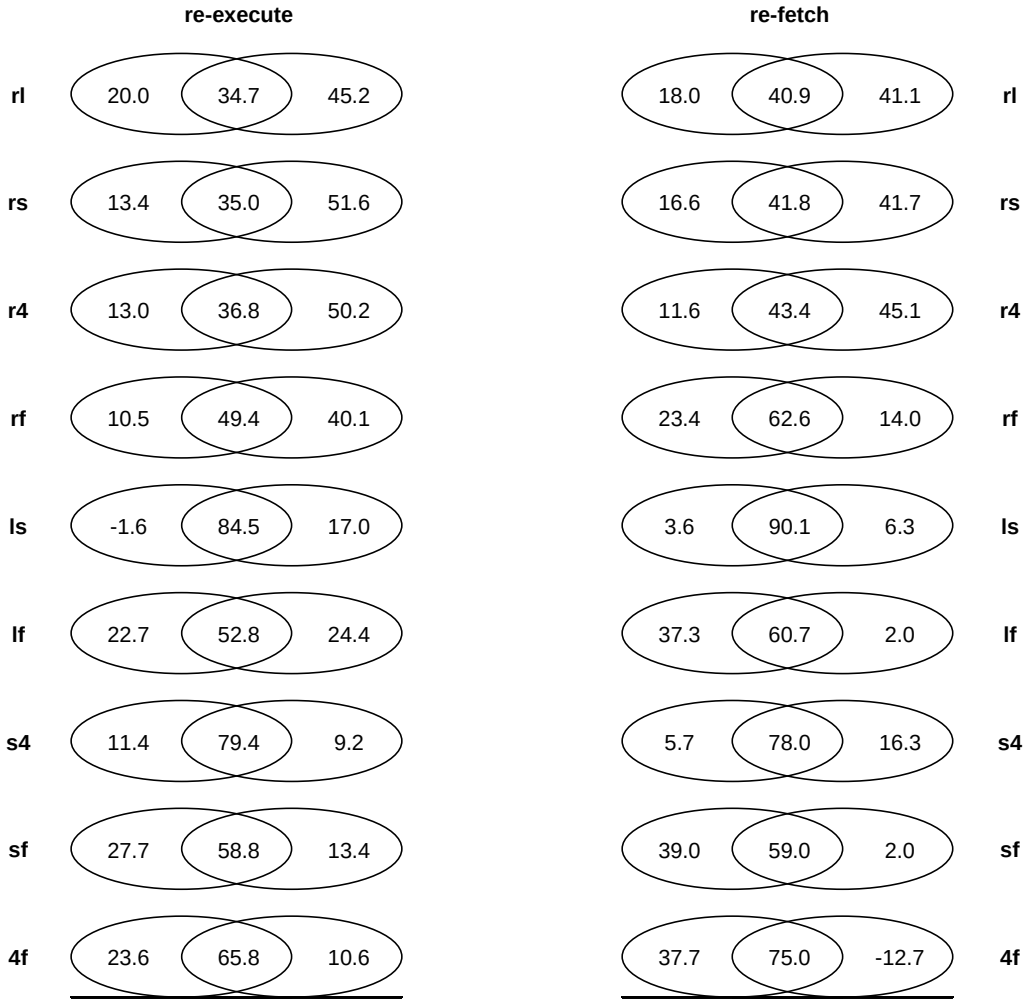

Figure 8.3: Re-execute and re-fetch Venn-diagrams for *SAg* hybrids.

The Reg+LV hybrid exhibits the smallest shared contribution of any two-component hybrid with both recovery mechanisms. Clearly, the Reg compo-

nent complements the LV component well and vice-versa, implying that each of them can predict important loads that the other one cannot. In fact, Reg complements any predictor well. The three predictors with the smallest overlap all include a Reg component. The Reg+St2d predictor, which was found to be the most effective two-component hybrid in the previous section, has the second smallest overlap both with re-fetch and re-execute. The three predictors with the smallest overlap are all among the best performing re-fetch hybrids. However, this observation does not apply to re-execute very well. Furthermore, for both recovery mechanisms there are hybrids that perform well in spite of a large shared contribution, for example, the St2d+L4V and St2d+FCM predictors.

The hybrids LV+St2d, St2d+L4V, and L4V+FCM exhibit a large shared contribution. In these three hybrids, both components predict mostly the same load instructions and therefore complement each other only poorly.

The LV component in the LV+St2d hybrid with re-execute and the FCM component in the L4V+FCM hybrid with re-fetch show a negative individual contribution because with re-execute the St2d component performs better than the LV+St2d hybrid and with re-fetch the L4V component outperforms the L4V+FCM hybrid. Therefore, hybridizing lowers the performance and thus results in a negative unique contribution in both cases.

The reason for this is the aforementioned negative interference between the involved components. Since single-component predictors do not require a selector whereas hybrids do, the culprit for the lower performance must be the imperfect selector. After all, the St2d component in the LV+St2d hybrid is identical to the single-component St2d predictor that outperforms the hybrid.

With only one exception (St2d+L4V), re-fetch recovery results in larger shared contributions than re-execute for the same predictors. This probably means that the easily predictable loads (i.e., loads that have very high confidences associated with them) tend to be the loads that can be predicted with either component. Those loads are most likely the runtime constants that are

always predictable because their values never change [CFE97].

Overall, the Reg predictor complements the other four predictors exceptionally well, meaning that it can predict a rather distinct set of load instructions. Second, but not nearly as good of a partner, is the FCM predictor. The St2d predictor does not complement the LV or the L4V predictor well. This is consistent with the results from Section 4.3.3, which show that there is only a small number of truly stride predictable load values (i.e., with a non-zero stride). The last value and the last four value predictor can both capture all the zero-stride predictable loads, eliminating the need for a stride predictor in those cases.

Figure 8.4 shows the speedup contributions for two *bimodal* predictors. The first one, St2d+L4V, is similar to the LD4V+St predictor proposed by Wang and Franklin [WaFr97], and the second one, St2d+FCM, is the hybrid proposed by Rychlik et al. [RFKS98] except it is not set-associative.

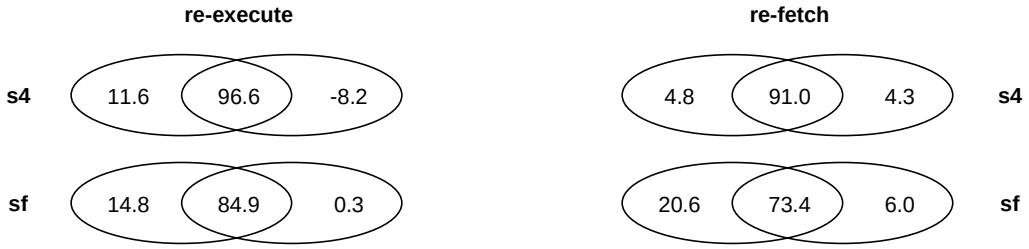

Figure 8.4: Re-execute and re-fetch Venn-diagrams for *bimodal* hybrids.

Figure 8.4 illustrates that the two predictors result in significantly more shared speedup with a *bimodal* CE than they do with a *SAg* CE (see Figure 8.3). In general, the shared contribution is quite high for both hybrids. In case of the St2d+L4V hybrid, all the last value predictable loads can be handled by both components, resulting in significant overlap. Something similar probably happens with the St2d+FCM hybrid because the FCM component can predict stride sequences after having seen them once. Furthermore, both the St2d and the FCM predictor are able to predict last value predictable

loads.

Together with the performance results from Figure 8.1 and Figure 8.2, we find that the substantially smaller and simpler Reg+St2d hybrid outperforms both the St2d+L4V and the St2d+FCM predictors with either misprediction recovery mechanism, illustrating the importance of component analyses when designing hybrid load value predictors.

8.3.2 Three-component Hybrids

Figure 8.5 shows the shared and unique speedup contributions (in percent of total predictor performance) of all *SAg*-based three-component hybrids. A set of seven equations has to be solved to compute the seven values in each Venn-diagram. The numbers in the center of each diagram denote the contribution that is shared among all three predictor components, the other three overlapping regions represent the shared contribution of all pairs of components, and the non-overlapped parts list the unique performance contributions. For example, in the top left (rs4) of Venn-diagram, the top left oval lists the contribution of the Reg component, the top right oval presents the contribution of the St2d component, and the oval at the bottom shows the contribution of the L4V component.

Figure 8.5 illustrates that all three components in the LV+St2d+FCM and the St2d+L4V+FCM hybrids suffer from large shared speedup. In both hybrids, over half of the performance is shared among all three components. The Reg+St2d+L4V and the Reg+LV+St2d predictors have somewhat large shared contributions, and the remaining hybrids exhibit relatively high unique contributions in at least one of their components. Note that Reg's unique contribution is at least seven percent in every case.

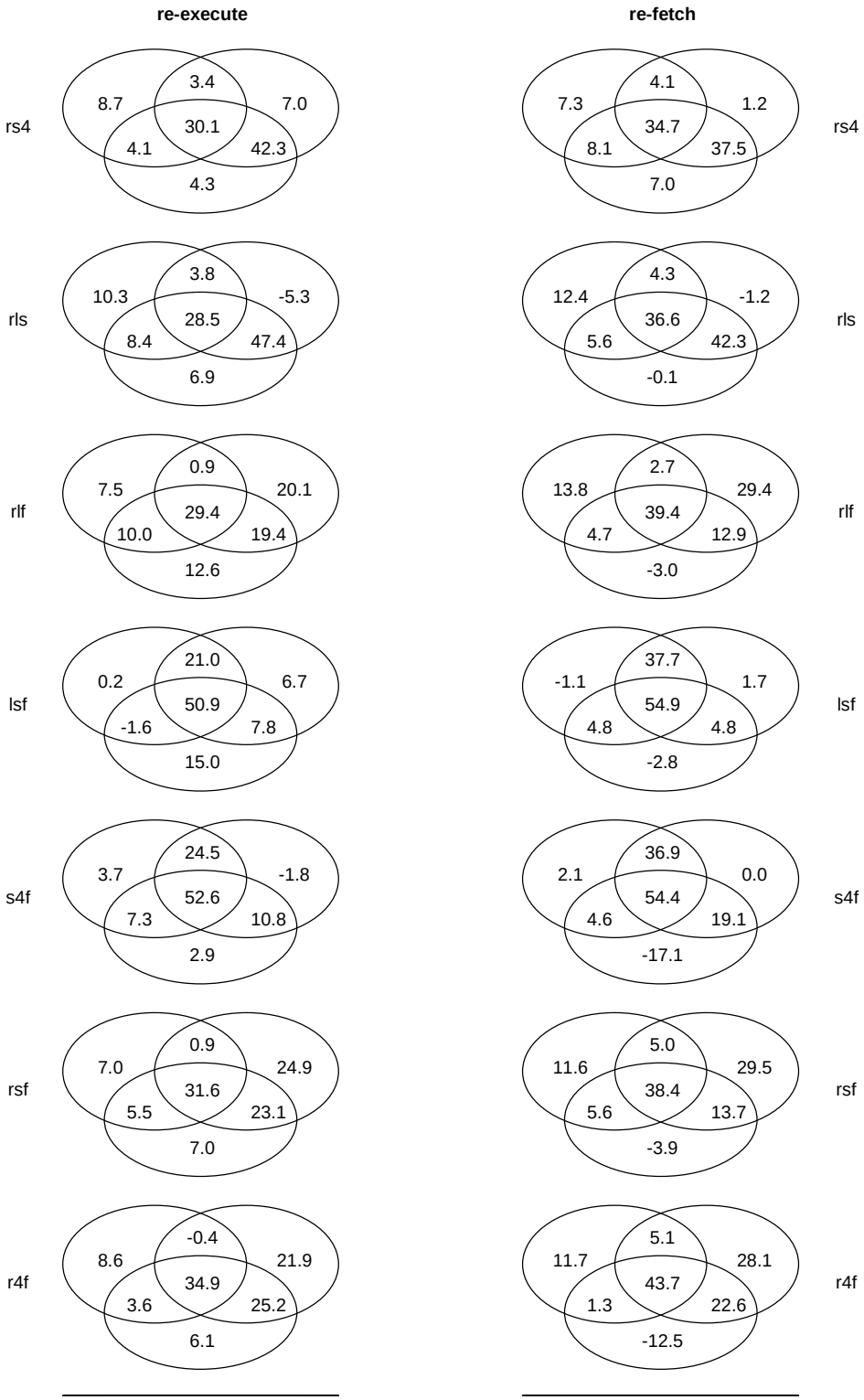

Figure 8.5: Venn-diagrams for *SAg*-based three-component hybrids.

As was true with the two-component predictors, the amount of sharing inversely correlates to a reasonable degree with the re-fetch performance of a hybrid, but there is not much correlation with the re-execute performance. Nevertheless, the Venn-diagrams illustrate nicely which components do not significantly contribute to the overall performance and can consequently be left out. More importantly, the diagrams reveal the components that cause negative interference and should therefore be removed.

Because it is hard to show a four-component Venn-diagram in two dimensions and because the four-component hybrids do not outperform the best three-component hybrids with re-fetch and do not significantly outperform the best three-component hybrids with re-execute, I will refrain from studying the speedup contributions of the two four-component hybrids.

8.4 Summary

This chapter presents the performance of all hybrid load value predictors that can be built out of the five basic predictors (Reg, LV, St2d, L4V, FCM). The results show that such hybrids are able to deliver substantially more performance than even the best single-component predictor because different components contribute independently to the overall performance.

Most hybrids, in particular the effective ones, perform better with a *SAg* confidence estimator than with a *bimodal* CE. Furthermore, there is evidence that the *SAg* CE embodies a better selector than the *bimodal* CE.

Studying the hybrids' performances as well as the unique and shared speedup contributions of their components revealed that the register predictor with its poor individual performance represents a valuable addition to all the other predictors. Conversely, combining well-performing predictors frequently does not result in an effective hybrid. In fact, some predictor combinations underperform a similar predictor with fewer components due to negative interference between the components. This can happen because adding compo-

nents to a predictor makes the task of the selector more difficult, which sometimes increases the selector-related losses more than the added predictability potential can compensate for. Hence, care must be taken when selecting predictors as components for a hybrid. To identify components that complement each other well, performance analyses are most likely unavoidable.

With re-fetch as well as when averaging the re-fetch and the re-execute speedups, the Reg+St2d+L4V predictor outperforms all the other studied predictors. The next chapter shows how this relatively large hybrid can be made smaller without compromising its effectiveness.

Chapter 9

Hybridizing with Hardware Reuse

Of all the predictors studied in Chapter 8, the Reg+St2d+L4V hybrid is the most promising load value predictor with re-fetch and one of the best performing ones with re-execute. Unfortunately, it is also rather large. The goal of this chapter is to shrink the size of this load value predictor without degrading its performance.

Several storage-reduction techniques are presented that together shrink the size of the Reg+St2d+L4V hybrid to less than half its original size without compromising its performance. Speedup comparisons show that this storage-reduced hybrid significantly outperforms other same-sized predictors from the literature. A sensitivity analysis concludes this chapter.

9.1 Shrinking the Reg+St2d+L4V Hybrid

Replacing the St2d with an St component in the Reg+St2d+L4V hybrid yields a first, very small size-reduction. The St component is smaller because it requires only one stride field whereas the St2d component requires two. Note that in Section 4.3.3 the stride 2-delta predictor was introduced because it outperforms the simpler stride predictor. As discussed, the performance difference between the two predictors is mainly a result of short sequences of repeating load values, which the stride predictor cannot handle well. However, a last value predictor can handle such sequences. Since a Reg+St+L4V hybrid already contains a last value predictor as part of the L4V component, it is not necessary for the stride component to be able to cope

with such sequences. Rather, the LV subcomponent will report a higher confidence than the St component for short repeating value sequences and will therefore be selected over the St component. As a result, the St component is only needed to predict the truly stride predictable loads, which it can do about as well as the St2d predictor. Consequently, there is no need for a St2d component because the smaller St component suffices.

If we assume ten-bit histories for the six *SAg* CEs in the Reg+St+L4V hybrid, 64 bits for retaining load values in the St and the L4V components, and eight-bit strides, then each predictor line in the Reg+St2d+L4V hybrid requires 396 bits of storage. Because the Reg+St+L4V hybrid has one fewer stride field, its lines are "only" 388 bits long, a saving of about two percent. Clearly, replacing the St2d with an St component does not significantly reduce the amount of state required by the predictor, but it does make the predictor a little less complex.

Note that switching the first two components (i.e., the Reg and the St component) in the Reg+St+L4V hybrid results in slightly improved performance (see Section 9.5.1), which is why an St+Reg+L4V hybrid will be used in the rest of this chapter.

9.1.1 Shrinking the L4V Component

The L4V predictor is by far the largest component in the St+Reg+L4V hybrid. Studying the four values stored in each line of this component revealed a relatively straight-forward way to compress them. As it turns out, the four values are almost always similar in magnitude. The reason for this is twofold. Data values often cluster around zero and do not generally use the whole range of numbers that can be represented with 64-bit values. Address values are also often similar. Their absolute values depend on where a program's data and code is mapped to in virtual memory, but memory is often allocated with significant spatial locality, which results in address values that are close

together.

Since the four values within each predictor line are similar in magnitude, their most significant bits are almost always identical. Hence, it suffices to store those bits only once instead of four times. Surprisingly, in the SPECint95 programs as many as 48 bits (or three quarters of all the value bits) can be shared among the four retained values virtually without degrading the predictor's performance [BuZo00].

Because the number of identical bits depends on the workload and the memory allocator, I designed a predictor in which "only" the 44 most significant bits are shared. The resulting predictor, which is called a last four *partial* value predictor (L4pV), still stores the full 64 bits of the most recently loaded (last) value in each line but retains only the twenty least significant bits of the second, third, and fourth last value. The four *SAg* CEs are unchanged. To form full 64-bit values, the twenty bits of the partial values are concatenated with the (shared) 44 most significant bits in the first component of the L4V predictor.

A 1024-line last four partial value predictor yields a harmonic mean speedup of 11.522% with re-fetch and 13.687% with re-execute on SPECint95. By comparison, the roughly same-sized L4V predictor with 512 lines delivers a re-fetch speedup of 10.978% and a re-execute speedup of 12.537%. This represents an improvement of one half to one percentage point over the L4V predictor, which has already been shown to perform best for the used predictor size of sixteen kilobytes for storing values (Chapter 7).

Note that the L4pV predictor has twice as many lines as the L4V predictor of approximately the same size. Among predictors with the same number of lines, the L4pV requires only about half as much state as the L4V predictor while basically delivering the same performance.

Replacing the L4V component in the St+Reg+L4V hybrid with an L4pV component with twenty-bit partial values reduces the line length from 388 bits to 256 bits, which amounts to a state reduction of 34% without compromising

the predictor's performance.

Initially, the St+Reg+L4pV hybrid included three valid bits in each line of the L4pV component that indicated whether the concatenation of the three partial values with the shared bits resulted in the correct 64-bit value. However, it turned out that valid bits are superfluous because the confidence estimators already perform the task of the valid bits. During predictor updates, an incorrect concatenation-result yields almost always a value that looks unpredictable. As a consequence, the CE disallows predictions just like a valid bit would. Therefore, the valid bits are omitted in the St+Reg+L4pV hybrid.

Instead of storing partial values, I also investigated storing 20-bit signed offsets. This approach is more complex because differences and sums have to be computed during each predictor access. Surprisingly, the resulting performance is lower than that of the simpler concatenation approach. Hence, partial values instead of offsets are used in the St+Reg+L4pV hybrid.

9.1.2 Making the Stride Predictor Storage-less

The second largest component in the St+Reg+L4pV hybrid is the stride predictor. In Wang and Franklin's LD4V+St predictor [WaFr97], the St component shares its 64-bit base value with the LD4V component. The same approach can be used in the St+Reg+L4pV hybrid because the L4pV and the St component both store the 64-bit last value (among other things). Hence, the St component's size can be reduced substantially by letting it use the last value stored in the L4pV component instead of having it retain its own copy. Note that sharing the last value has no effect on the performance but reduces the predictor size.

Interestingly, the St component's stride field is also superfluous. The stride records the difference between the last loaded value and the second to last loaded value. Since the L4pV component retains both the last and the second to last loaded value, the difference can be computed on-the-fly, elimi-

nating the need to store it explicitly [BuZo00]. The predicted value evaluates to the second last value subtracted from the last value multiplied by two (i.e., shifted to the right by one bit).

As a result, the whole stride component has become storage-less, just like the register component. Both components consist of only a confidence estimator and obtain the values for making predictions from the register file and the L4pV component.

The St+Reg+L4pV hybrid with a storage-less stride component requires 184 bits of state per predictor line. This represents a saving of 28% compared to the 256 bits that the St+Reg+L4pV hybrid with a regular stride component requires.

Section 9.5.3 shows that the fourth subcomponent in the L4pV predictor does not add to the St+Reg+L4pV hybrid's performance. It can (and should) therefore be left out, which further reduces the predictor size and complexity.

The resulting St+Reg+L3pV predictor with its storage-less stride, storage-less register, and a storage-reduced last three value component requires a total of 154 bits of storage per predictor line. As compared to the initial Reg+St2d+L4V predictor with its 396-bit lines, this represents a saving of 61% of state in each line. The saving over the entire predictor is somewhat smaller because the size of the second-level of the *SAg* CEs remains constant. For example, the storage-reduced St+Reg+L3pV hybrid with 512 lines is 56% smaller than the initial Reg+St2d+L4V hybrid with the same number of lines.

Note that the size of this predictor has been reduced to less than a half almost without loss of performance. For example, the Reg+St2d+L4V hybrid's *SAg* speedup with re-fetch is 13.558% and with re-execute it is 17.189%. Using the exact same CE setting and the same number of predictor lines, the two times smaller storage-reduced St+Reg+L3pV hybrid yields a re-fetch speedup of 13.544% and a re-execute speedup of 16.825%.

9.2 Coalescing Hybrid Predictor Components

In the previous section, the size of all the components except the register component was reduced. Since the Reg predictor essentially only consists of a confidence estimator, a technique to shrink CEs is necessary to make the Reg predictor smaller. Finding approaches to reduce the size of confidence estimators is left for future work.

In the previous section, the Reg+St2d+L4V hybrid was used as the basis for the size-reduction approaches because this predictor performs exceptionally well. However, it is of course also possible to reduce the size of other hybrids. For example, a stride predictor can be made storage-less in connection with any last *n* value predictor that has a width of at least two. In an St+LV hybrid, the LV component can be made storage-less by reusing the last value information from the St component. Hybridizing an St or LnV predictor with an FCM predictor also allows some state sharing. The first level of the FCM stores the last *n* values in a hashed form. If the hash can be computed on-the-fly, the hashed result of the last value (in case of an St component) or the last *n* values (in connection with an LnV predictor) need not be stored because the necessary information to compute it can be obtained from the non-FCM component.

9.3 The Coalesced-Hybrid

To further improve the performance, I added one more enhancement to the St+Reg+L3pV predictor. Bekerman et al. [BJR+99] and, independently, by Calder et al. [CRT99] found that infrequently executed loads that alias with frequently executed loads evict useful predictor entries often enough to degrade the performance. According to their suggestion, I added one bit to the partial tags (which I termed *b-tags*) to indicate whether the last access to a given predictor line resulted in a tag miss. This bit makes it possible to pre-

vent updating a predictor line after the first tag miss. Only allowing updates after two or more misses in a row effectively prevents infrequently executed loads from being able to pollute the predictor. The extra bit in the b-tags essentially represents a one-bit replacement counter [CRT99].

I named the resulting b-tagged, *SAg*-based St+Reg+L3pV predictor *coalesced-hybrid* because its components and subcomponents are fused together by sharing a large amounts of state.

Figure 9.1 shows the architecture of the coalesced-hybrid load value predictor with its storage-less stride, storage-less register, and reduced-storage last three partial value component. The numbers in the fields indicate their widths in number of bits.

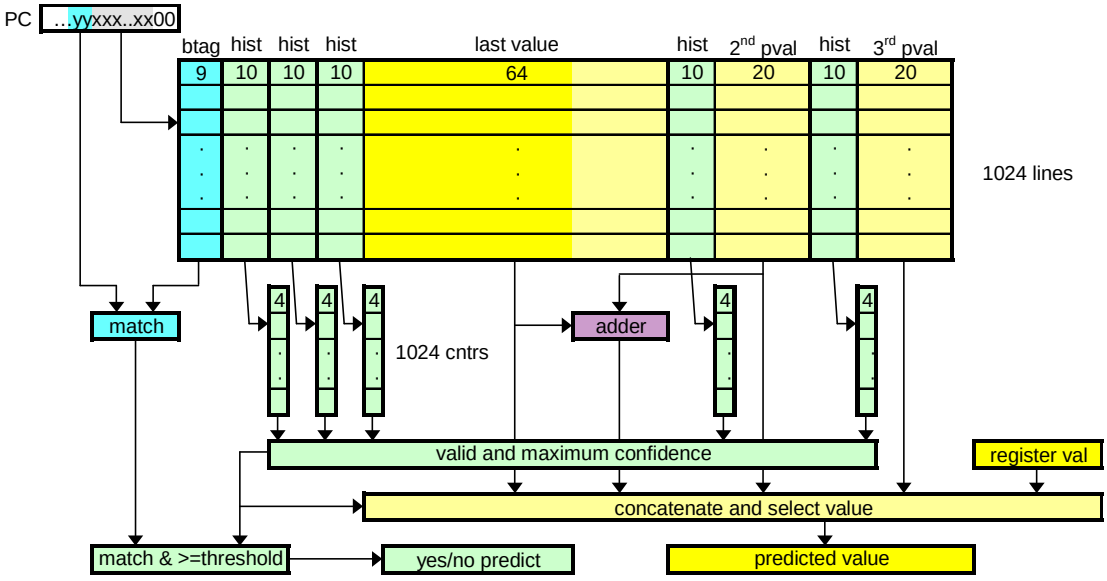

Figure 9.1: The architecture of the coalesced-hybrid load value predictor.

Every line of the predictor includes a nine-bit partial b-tag. The first of the five identical *SAg* confidence estimators (they each consist of an array of ten-bit histories “hist” and an array of three or four-bit saturating counters) forms the storage-less register value component (Reg), which uses values from the CPU’s register file for making predictions. The second confidence estimator

belongs to the stride storage-less predictor (St) whose only other element is the adder since it uses values from the L3pV component for making predictions. The remaining three confidence estimators, the 64-bit and the two twenty bit value fields form the last three partial value (L3pV) component.

The five sub-components operate independently and perform five value predictions and five confidence estimations in parallel. The value of the component reporting the highest confidence is used for making a prediction, but only if the confidence is above the preset threshold. To break ties, the components are prioritized from left to right, that is, the stride component has the highest priority, the register component has a medium priority, and the last three value component has the lowest priority. Within the last three value component, the more recent values have a higher priority.

When the predictor is updated, each component again makes a value prediction whose result is compared with the true load value. The confidence estimators are then updated based on the outcome of this comparison. At the same time, the values within the *L3pV* component are passed on to the next "older" sub-component and the true load value is copied into the 64-bit last value field.

9.4 Coalesced-Hybrid Performance

This section compares the performance of the coalesced-hybrid with other load value predictors from the literature as well as with oracle predictors.

9.4.1 Comparison with Other Predictors

This section compares the harmonic-mean speedups over SPECint95 of several well-performing predictors from the literature and some of mine. The seven predictors I consider are:

- **Tag *Bim* LV**, a partially tagged *bimodal* last value predictor [LWS96]

  This predictor has one of the least complex architectures.

- **Tag *SAg* L4V**, a partially tagged *SAg* last four value predictor [BuZo99b]

  This is the predictor from Chapter 7.

- **Tag *SAg* St2d**, a partially tagged *SAg* stride 2-delta predictor [SaSm97a]

  This is Sazeides and Smith's stride 2-delta predictor augmented with my *SAg* confidence estimator, resulting in one of the best-performing single-component load value predictors.

- **Tag *Bim* St2d+FCM**, a partially tagged *bimodal* stride 2-delta and finite context method hybrid [RFKS98]

- **B-Tag *SAg* St+Reg+L3pV**, the partially b-tagged *SAg* coalesced-hybrid predictor [BuZo00]

  This hybrid includes a storage-less stride, a storage-less register, and a reduced-storage last three (partial) value component. The b-tags include a one-bit replacement counter.

- **Tag *apBim* LD4V**, a tagged last distinct four value predictor with an access-pattern-based *bimodal* confidence estimator [WaFr97]

- **Tag *apBim* LD4V+St**, a hybrid between the LD4V and a *bimodal* stride predictor [WaFr97]

  This hybrid uses a two-bit *bimodal* selector. The St component gets its base value from the LD4V component, i.e., the two components are coalesced.

Since the predictors vary greatly in their architectures and complexities, they cannot be scaled to be of identical size. Consequently, it is only possible to compare predictors of similar sizes. I was able to scale all seven predictors to require between 19 and 31 kilobytes of state, which I believe is a realistic size for a load value predictor to be included in an upcoming CPU. From these base-configurations I created two additional configurations for each predictor, a smaller one (by quartering the number of predictor lines) and a larger one (by quadrupling the number of predictor lines). The resulting predictor sizes for the three size-ranges (denoted as small, base, and large) are shown in Table 9.1. Note that the numbers in the table include the amount of state required to retain values, tags, and confidence information and are for a re-fetch architecture. With re-execute, some of the predictors require a little less state because the saturating counters are smaller. Because the access-pattern-based *bimodal* CE (*acBim*) in the LD4V predictor as well as the FCM predictor require large second-level tables, splitting predictors that include either an FCM or an LD4V component into several banks would substantially increase their size. The amounts shown in the table are for non-banked FCM and LD4V sizes.

Predictor Size in Kilobytes of State			
	small	base	large
Tag Bim LV	4.8	19.0	76.0
Tag acBim LD4V	12.3	25.0	76.0
Tag acBim LD4V+St	12.4	25.6	78.5
Tag SAg St2d	8.1	26.5	100.0
Tag SAg L4V	12.8	27.0	84.0
B-Tag SAg St+Reg+L3pV	15.1	30.4	91.5
Tag Bim St2d+FCM	19.9	31.5	78.0

Table 9.1: State requirement of the seven predictors' three configurations.

A detailed parameter space evaluation was performed to determine the CE setting that yields the highest speedup for each of the seven predictors. This evaluation included varying the prediction threshold and the counter decrement (penalty). The counter-top is fixed at sixteen for re-fetch and eight

rement (penalty). The counter-top is fixed at sixteen for re-fetch and eight for re-execute unless otherwise noted. The St2d+FCM hybrid allows many different ways of distributing the state over its two components. The second-level table of the FCM component used in this study has a capacity of 2048 entries. Table 9.2 shows the parameters of the best base-configurations of the seven predictors.

	predictor	tag	hist	re-execute			re-fetch		
	lines	bits	bits	top	thr	pen	top	thr	pen
Tag Bim LV	2048	8	-	8	5	1	16	10	15
Tag acBim LD4V	512	8	-	8	3	2	16	14	9
Tag acBim LD4V+St	512	8	-	8	7	2	16	13	10
Tag SAg St2d	2048	8	10	8	5	1	16	12	5
Tag SAg L4V	512	8	10	8	7	4	16	14	11
B-Tag SAg St+Reg+L3pV	1024	8+1	10	8	7	2	16	14	9
Tag Bim St2d+FCM	1024/2048	8	-	8	5	1	16	15	11

Table 9.2: The base-configurations of the seven predictors.

All the predictors are configured to work as well as possible in their base-configuration (19 to 31 kilobytes of state). Except for the number of predictor lines, the same parameters are used with the other two predictor sizes and no search for an optimal setting is performed. I use this approach to mimic what would happen if programs that are much larger or much smaller than the SPECint95 programs are run on these predictors. The intuition is that a larger program performs similarly on a load value predictor to a smaller program on a proportionately smaller version of the same predictor.

Figure 9.2 and Figure 9.3 present the harmonic-mean speedups of the eight predictors with a re-fetch and a re-execute misprediction recovery mechanism, respectively. Three speedup results are shown for each predictor corresponding to the three predictor sizes.

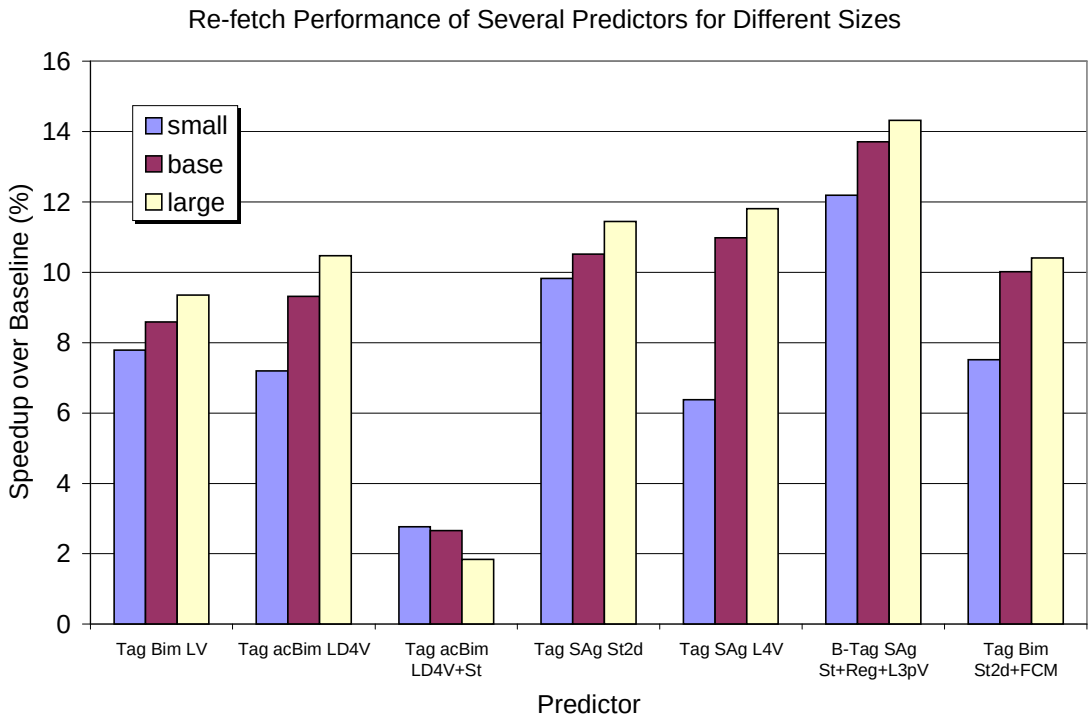

Figure 9.2: Re-fetch speedup of several predictors for three sizes.

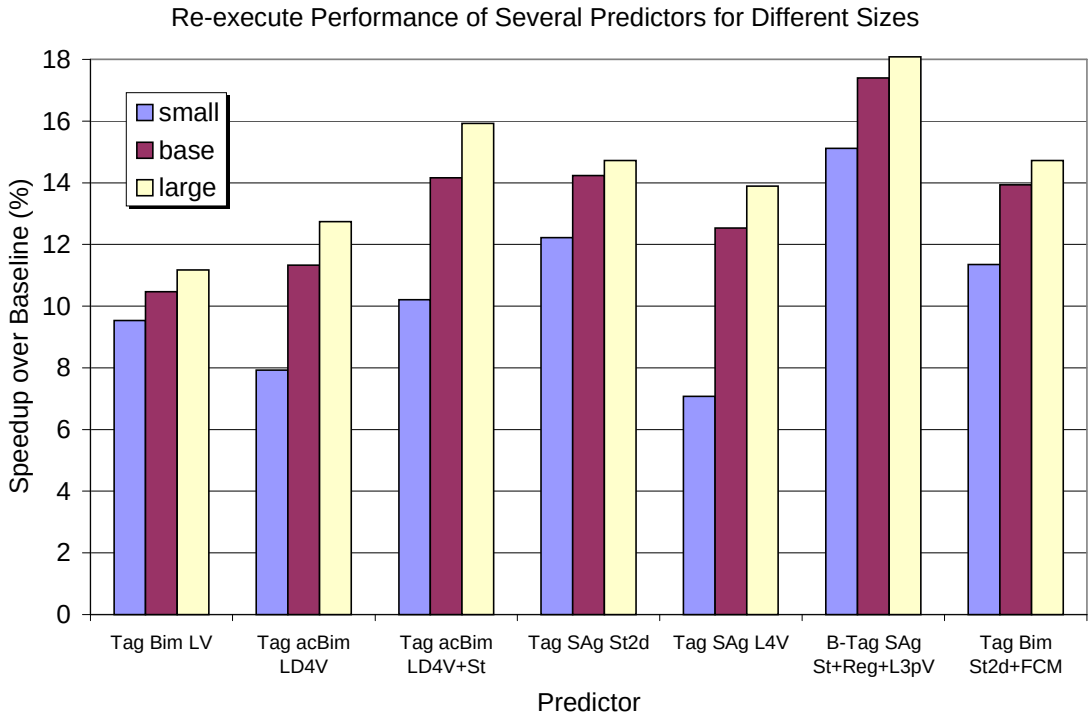

Figure 9.3: Re-execute speedup of several predictors for three sizes.

The coalesced-hybrid predictor (B-Tag *SAg* St+Reg+L3pV) clearly outperforms the other predictors both with a re-fetch and a re-execute misprediction recovery policy. More importantly, its *re-fetch* speedup is close to the other predictors' *re-execute* speedup in the larger size-ranges and actually exceeds it in the smallest size-range.

Furthermore, the performance of the smallest coalesced-hybrid configuration (which requires fifteen kilobytes of state) surpasses the performance of the other predictors, including the ones from the largest size-range that require over five times as much state. Only the Tag *acBim* LD4V+St predictor is able to slightly outperform the five times smaller coalesced-hybrid when re-execute is used. This clearly shows that hybrid predictors do not necessarily have to be large to perform well and that coalescing the components in a hybrid predictor is a very effective technique to save state.

The good re-fetch speedup of the coalesced-hybrid is encouraging, in particular because it allows microprocessor designers to use the already existing branch misprediction hardware to recover from load value mispredictions, which makes it less urgent to design and add a processor core that is capable of re-execution.

Note that the performance of the Tag *acBim* LD4V+St predictor actually decreases with re-fetch when increasing the predictor size. An investigation of this phenomenon revealed a somewhat surprising result. As it turns out, the smallest configuration of this predictor suffers significantly from aliasing. The confidence estimator detects this problem and inhibits the affected lines from making predictions. Consequently, the predictor only attempts relatively few predictions, which is reflected in its low performance when compared to the other predictors. The larger versions of this predictor suffer less from aliasing and the confidence estimator allows more predictions to take place. Unfortunately, the CE also allows significantly more incorrect predictions, which more than offset the benefit of the additional correct predictions. Hence, the overall performance decreases as the predictor becomes larger.

Using a better selector and confidence estimator than the suggested two-bit bimodal CE [WaFr97] would most likely increase this predictor's re-fetch performance.

Interestingly, the stride 2-delta predictor (Tag *SAg* St2d) performs better than the Tag *Bim* St2d+FCM hybrid. The reason is partly the difference in the confidence estimators and partly the fact that the relatively small FCM component does not perform very well and therefore takes away valuable real-estate from the St2d component. With even larger predictor sizes, the Tag *Bim* St2d+FCM hybrid would probably surpass the Tag *SAg* St2d predictor's performance.

Among the predictors in a given size-range, the predictors with more components have fewer lines (i.e., are shorter) than the single-component predictors and are consequently more likely to experience capacity problems, in particular in the smallest configuration. The effect of the resulting aliasing is visible in the two figures. The performance difference between the small and the middle configuration is significantly larger with the multi-component predictors (Tag *SAg* L4V, Tag *acBim* LD4V, and Tag *acBim* LD4V+St) than with the other predictors. Note that the coalesced-hybrid has more components than the Tag *SAg* L4V and the Tag *acBim* LD4V predictors, yet it is not affected nearly as much by detrimental aliasing since the high degree of coalescing allows it to have twice the number of predictor lines, which alleviates the capacity problem.

9.4.2 Comparison with Oracles

In Section 7.4.1 the last four value predictor was compared to versions of itself that contain oracles. In this section the same is done with the coalesced-hybrid to determine how much larger a fraction of the existing potential this predictor can reap.

The first predictor (*no-oracle*) represents the coalesced-hybrid in its con-

ventional and implementable form as it is described in Section 9.3. It does not include an oracle. The first oracle (*ce-oracle*) represents the same predictor except it incorporates a perfect confidence estimator. The next oracle (*ce/sel-oracle*) is the coalesced-hybrid with a perfect confidence estimator and a perfect selector. The *all(b-tag)-oracle* makes a correct load value prediction whenever there is a tag-match in the 1024-line b-tags. The final oracle (*all-oracle*) predicts every executed load with the correct value.

None of the genuine oracles make any mispredictions, but the *ce-oracle* makes imperfect selections. Only the *all-oracle* always makes a prediction. Figure 7.4 shows the speedups delivered by the oracle-less predictor and the four oracles.

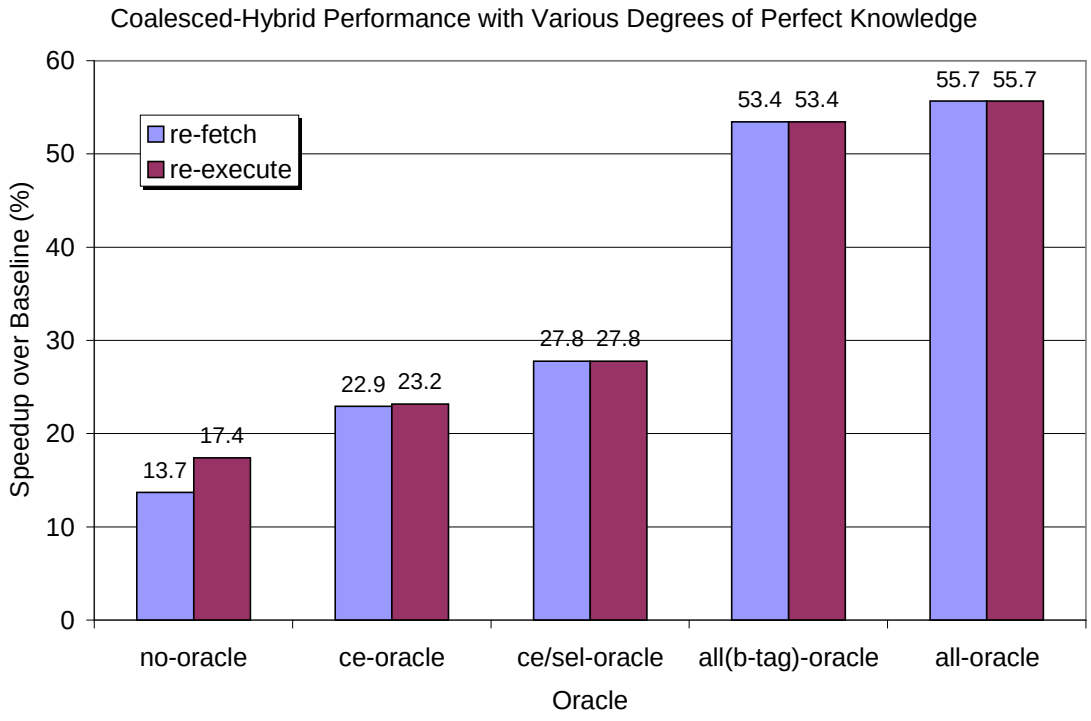

Figure 9.4: Performance of different coalesced-hybrid oracles.

The figure shows that neither the confidence estimator nor the selector perform close to optimal in the *no-oracle* predictor. Again, because there are

no mispredictions, the discrepancy between the re-fetch and the re-execute speedup with the *ce-oracle* must stem from the dissimilar CE settings that affect the performance of the selector. The difference is more pronounced with the coalesced-hybrid than it is with the last four value predictor (see Section 7.4.1). The reason could be the more regular design of the L4V predictor, which may make the task of the selector easier and less susceptible to the CE setting than in the Reg+St+L3pV hybrid.

Overall, the coalesced-hybrid's confidence estimator and selector (*no-oracle*) are able to reap 49% to 63% of the theoretically possible speedup (*ce/sel-oracle*) for this predictor. A comparison with the perfect load value predictor (*all-oracle*), however, shows that the predictor only yields 25% to 31% of the speedup that can theoretically be attained with load value prediction (*all-oracle*). Comparing *all-oracle* with *ce/sel-oracle* shows that the coalesced-hybrid only contains the necessary information to reach half the possible speedup.

Finally, doubling the predictor height (due to savings in the predictor width) and adding b-tags considerably improves the potential of the predictor because the *all(b-tag)-oracle* delivers 96% of the *all-oracle*'s performance, whereas the *all(tag)-oracle* from Figure 7.4 only yields 92% of the performance.

9.5 Coalesced-Hybrid Sensitivity Analysis

This section investigates the sensitivity of the coalesced-hybrid's performance to the order of its components, the type of tags used, and the width of the last *n* partial value component.

9.5.1 Component Permutations

The five sub-components that make up the coalesced-hybrid load value predictor are prioritized from left to right, that is, the stride component has the highest priority, the register component has a medium priority, and the last three value component has the lowest priority. Within the last three value component, the younger a value the higher its priority, which is slightly beneficial as was determined in a previous study [BuZo99b]. The prioritization is used to resolve ties between multiple components reporting the same highest confidence. If only one component reports the highest confidence, then it is selected independent of its priority.

Table 9.3 shows the re-fetch and the re-execute speedup of the coalesced-hybrid when its three major components are permuted, i.e., when the prioritization is changed. The predictors shown in the table are sorted from top to bottom by decreasing average performance.

Permutation Speedups		
	re-fetch	re-execute
St+Reg+L3pV	13.711	17.400
Reg+L3pV+St	13.666	17.419
Reg+St+L3pV	13.577	17.318
L3pV+Reg+St	13.579	17.182
L3pV+St+Reg	13.470	17.189
St+L3pV+Reg	13.461	16.971

Table 9.3: Speedup of the six main component permutations.

Clearly, no order is significantly better that any other. All the permutations yield speedups within three percent of one another. The only minute correlation I was able to detect is that prioritizing the L3pV component over the Reg component seems to be slightly disadvantageous. I use St+Reg+L3pV because it performs best by a slight margin in the re-fetch case and also when averaging the re-fetch and the re-execute numbers.

9.5.2 Tags and B-Tags

The coalesced-hybrid includes a special kind of (partial) tags that I call b-tags. B-tags contain one extra bit that indicates whether the last access to any given line in the predictor resulted in a tag miss. This information is used to inhibit updates from evicting the contents of a predictor line until two or more misses have been seen in a row. As a consequence, infrequently executed loads cannot easily expel information about frequently executed loads that they alias with, which improves the predictor performance. The extra bit essentially represents a one-bit replacement counter [CRT99].

Figure 9.5 shows how the coalesced-hybrid performs with eight-bit partial tags, eight-bit partial b-tags, and full b-tags.

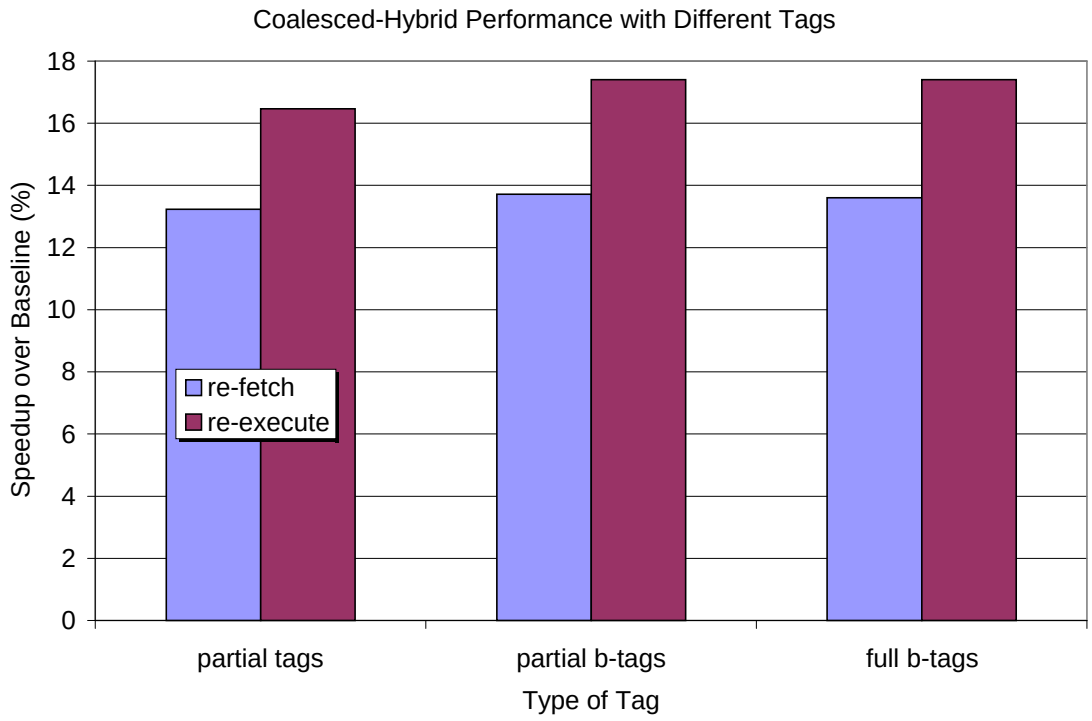

Figure 9.5: The coalesced-hybrid's speedup with various tag schemes.

There is hardly any difference between the performance with partial and

full b-tags. The reason is that eight bits are sufficient to allow the predictor to distinguish between all the load instructions in seven of the eight benchmark programs. Only *gcc*'s executable is too large. While the seven programs perform exactly the same with eight-bit partial b-tags and full b-tags, *gcc* actually performs slightly better when only partial b-tags are used, in particular with re-fetch. Clearly, the resulting aliasing must somehow be helpful. Nevertheless, I believe that in most cases larger executables require more tag-bits for optimal performance.

Using b-tags instead of normal tags improves the speedup of the coalesced-hybrid by half a percent with re-fetch and close to one percent with re-execute. Evidently, infrequently executed load instructions do affect the coalesced-hybrid's ability to correctly predict the frequently executed loads that they alias with.

9.5.3 Predictor Width

Figure 9.6 is presented to determine how wide the last *n* partial value component in the coalesced-hybrid needs to be. It shows the predictor's performance with differently sized, twenty-bit last *n* partial value components.

The performance of all the predictors in the figure is quite similar. Only the St+Reg+L2pV predictor's speedup is somewhat lower. Clearly, retaining three last values per line appears to be sufficient to reap almost all the potential. This result is particularly surprising because the predictors shown in the figure are not scaled to the same size but become larger as the width increases. Consequently, I use a last three partial value component in the coalesced-hybrid to keep the predictor's size and the number of components small while still reaping most of the performance.

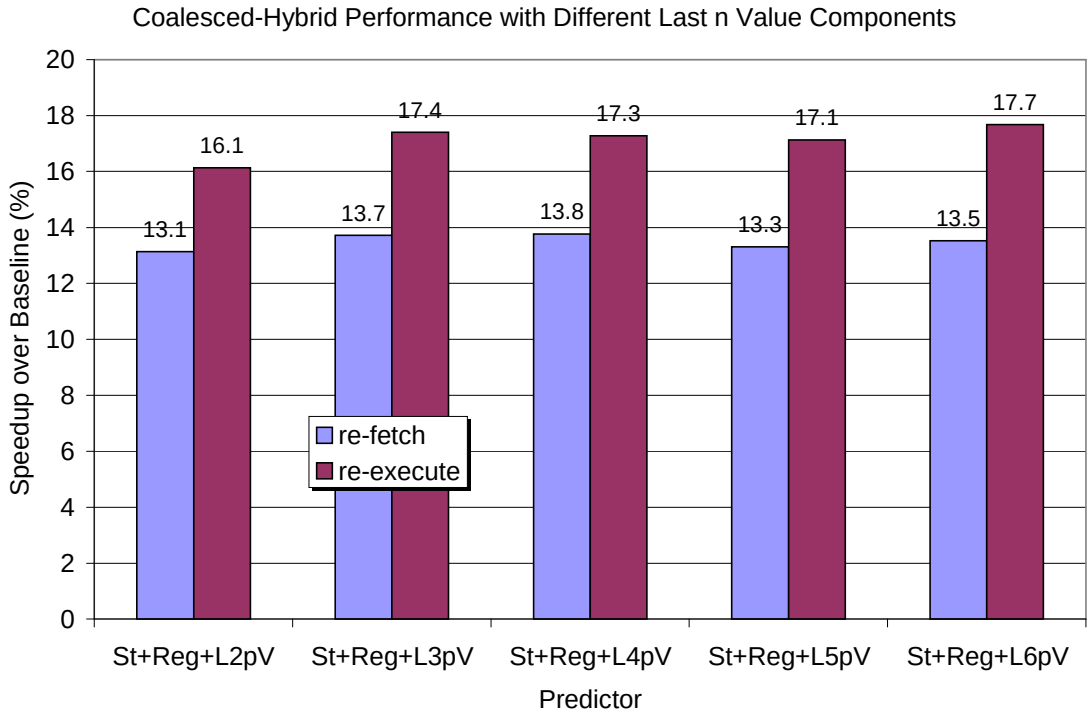

Figure 9.6: Performance with different last *n* partial value components.

Note that this result indicates that hybridization is more important than making the hybrid's components wider to improve their individual performance (Chapter 7).

Interestingly, the re-fetch performance drops when going from a last four to a last five partial value component and the re-execute speedup decreases when going from a last three to a last four and then again to a last five partial value component before increasing again. Since an St+Reg+L(n+1)pV predictor contains a complete St+Reg+LnpV predictor, these performance fluctuations once again have to be the result of increased negative interference between the growing number of predictor components.

9.6 Summary

This chapter describes several state-reduction techniques to decrease the often large storage requirement of hybrid load value predictors. For example, the amount of state required by a last *n* value predictor and a stride predictor can be reduced by a factor of two or more. This substantial saving can be achieved by having the last *n* value component provide all the information that the stride component needs, making the latter storage-less. In addition to that, the size of the last *n* value predictor can be decreased by sharing most of the bits among the *n* values in each predictor line. Both techniques result in a large reduction in predictor size while essentially maintaining the predictor performance.

Based on the well-performing Reg+St2d+L4V hybrid, I designed a very effective *coalesced-hybrid* load value predictor that requires only a small amount of state because it incorporates a storage-less stride, a storage-less register, and a reduced-storage last three partial value predictor. Cycle-accurate pipeline-level simulations of a four-way superscalar out-of-order CPU with many different predictors from the literature show that the coalesced-hybrid outperforms even the best predictors by almost twenty percent

over a wide range of predictor sizes. In the smallest configuration I investigated, which requires fifteen kilobytes of state, the coalesced-hybrid yields a speedup with both a re-fetch and a re-execute misprediction recovery mechanism that surpasses the speedup of other predictors, including predictors that are five times larger.

Chapter 10

Related Work

This chapter describes ideas from the current load value prediction literature, most of which are used or built upon in this dissertation, and discusses some of the differences to the presented work.

10.1 Early Work

Two independent research efforts [Gab96, LWS96] first recognized that load instructions exhibit *value locality* and concluded that there is potential for prediction.

Lipasti et al. [LWS96] investigated why load values are often predictable and how predictable different kinds of load instructions are. They found that while all types of loads exhibit significant value predictability, address loads have slightly better value locality than data loads, instruction address loads hold an edge over data address loads, and integer data values are more predictable than floating-point data values.

In a follow-up paper, Lipasti and Shen [LiSh96] broaden their scope to predicting all result generating instructions and show how value prediction can be used to exceed the existing instruction level parallelism (ILP) by collapsing dependencies. They found that using a value predictor delivers three to four times more speedup than doubling the data cache (same hardware increase) and argue that a value predictor is unlikely to have an adverse effect on the processor cycle time whereas doubling the data-cache size probably would. Furthermore, they note that loads are the most predictable

frequently executed instructions. Lipasti and Shen also propose the re-execute misprediction recovery mechanism, which is better suited for value prediction than the more conservative re-fetch recovery mechanism from the branch prediction literature. Both mechanisms are discussed in Section 2.2.

Gabbay's dissertation proposal [Gab96] also discusses general value prediction and how to boost the ILP beyond the data-flow limit, but he studies load value prediction by itself as well.

10.2 Predictors

Lipasti et al. [LWS96] describe a last value predictor to exploit the existing load value locality. Their predictor utilizes two-bit saturating up/down counters to classify loads as unpredictable, predictable, or constant, i.e., the saturating counters essentially represent a *bimodal* confidence estimator. This kind of confidence estimator is discussed and improved upon in Chapter 5.

Gabbay [Gab96] proposes four predictor schemes: a tagged last value predictor, a tagged stride predictor, a register-file predictor, and a sign-exponent-fraction (SEF) predictor. The SEF predictor is useful for predicting IEEE floating-point loads, and the register-file predictor was later improved by Tullsen and Seng [TuSe99].

In their next paper, Lipasti and Shen [LiSh96] suggest making predictions based on the last *n* values instead of the last value. However, they only provide results for oracle predictors. Chapter 7 presents an implementable last *n* value predictor.

Wang and Franklin [WaFr97] propose a two-level prediction scheme that makes predictions based on the last distinct four loaded values. They further propose a hybrid predictor that combines their last distinct four value predictor with a stride predictor. The two components in their hybrid are somewhat coalesced.

Sazeides and Smith [SaSm97a] perform a theoretical limit study of the

predictability of data values. They investigate the performance of three models: last value, stride 2-delta, and finite context method. Their finite context method predictor predicts the next value based on a finite number of preceding values by recording which value followed which sequence of values in the past. In a follow-up paper [SaSm97b], Sazeides and Smith design an implementable two-level value predictor based on the finite context method. They found that their predictor outperforms other, simpler predictors, but only with large predictor sizes. Rychlik et al. [RFKS98] propose a hybrid between Sazeides and Smith's finite context method and stride 2-delta predictors.

Tullsen and Seng [TuSe99] present a register value predictor that is storage-less except for its confidence estimator. It predicts that a load will fetch a value that is identical to the value already stored in the target register of the load instruction. Since the predictor uses the CPU's register file as a source for values, it does not require any value storage. I found the register value predictor to be a valuable complement in hybrid predictors (Chapter 8). I further show that a stride predictor can also be made storage-less in combination with a last two value predictor (Chapter 9).

Most of the above predictors and many more are discussed and compared performance-wise in Section 5.4 and Chapter 8.

10.3 Profile-based Approaches

Gabbay and Mendelson [GaMe97] explore the possibility of using program profiles to enhance the efficiency of value prediction. They use profiling to insert opcode directives to filter out highly unpredictable values from being allocated in the load value predictor, which considerably reduces the amount of aliasing. However, not even manual fine-tuning of the user supplied threshold value allows them to outperform their relatively basic hardware-only predictor in all cases. Furthermore, they found that training runs generally correlate with test runs, indicating that a program's input values do not signifi-

cantly affect the value predictability.

Calder et al. [CaFe99, CFE97] examine the invariance found from profiling instruction values and propose a new type of profiling called *convergent profiling*, which is much faster than conventional profiling. Their measurements reveal that a significant number of instructions (including loads) generate only one value with high probability. They note that the invariance of load values is crucial for the prediction of other types of instructions (by propagation). Similar to Gabbay and Mendelson's result, Calder et al. also found that the observed value invariance does not change significantly across different sets of program inputs.

Unfortunately, it is often difficult to obtain (good) profile information and extra bits for flagging instructions are usually not available in existing instruction sets, which is why profile-based approaches are excluded from this dissertation.

10.4 Other Related Work

Rychlik et al. [RFKS98] address the problem of useless predictions. They introduce a simple hardware mechanism that inhibits predictions that were never used (because the true load value became available before the predicted value was consumed) from updating the predictor, which results in improved performance due to reduced predictor pollution. Unfortunately, incorporating their scheme is not possible with the *SAg* confidence estimator (CE) that my predictors are based on. Finding and studying a similar mechanism that works with a *SAg* CE is left for future work.

In their next paper [GaMe98], Gabbay and Mendelson show that the instruction fetch bandwidth has a significant impact on the efficiency of value prediction. They found that value prediction (of one-cycle latency instructions) only makes sense if the producer and consumer instructions are fetched during the same cycle. Hence, general value prediction is more ef-

fective with high-bandwidth instruction fetch and issue mechanisms. They argue that current processors can effectively exploit less than half of the correct value predictions, since the average true data-dependence distance is greater than today's fetch-bandwidth of four. This is one of the reasons why this thesis focuses on predicting only load values, which requires significantly smaller and simpler predictors while still reaping most of the performance potential.

Gonzalez and Gonzalez [GoGo98] found that the benefit of data value prediction increases significantly as the instruction window size grows, indicating that value prediction is likely to play an important role in future processors. Furthermore, they observed an almost linear correlation between the number of correctly predicted instructions and the resulting performance improvement.

Fu et al. [FJLC98] propose a mixed hardware and software-based approach to value speculation that leverages advantages of both hardware schemes for value prediction and compiler schemes for exposing instruction-level parallelism. They propose adding new instructions to explicitly load values from the predictor and to update the predictor. In this dissertation, I limit myself to investigating only transparent prediction schemes, that is, predictors that do not require changes to the instruction set architecture and that can therefore be included in existing CPU families.

A more detailed study about predictability by Sazeides and Smith [SaSm98] illustrates that most of the predictability originates in the program control structure and immediate values, which explains the frequently observed independence of program input. Another result of their work is that over half of the mispredicted branches actually have predictable input values, implying that a side-effect of value prediction should be improved branch prediction accuracy. Gonzalez and Gonzalez [GoGo98] did indeed observe such an improvement in their study and propose possible predictor implementations to exploit it in a follow-up paper [GoGo99].

Sodani and Sohi's paper [SoSo98] builds on the Gonzalez studies. They found, among other things, that resolving branches using predicted operands is only beneficial in the presence of low value misprediction rates.

Rychlik et al. [RFKS98] and Reinman and Calder [ReCa98] propose reusing the confidence estimators in the components of their hybrid load value predictor as selector, thus eliminating the need for extra storage to guide the selection process. I use the same approach in my hybrid predictors.

Calder et al. [CRT99] examine selection techniques to minimize predictor capacity conflicts by prohibiting unimportant instructions from using the predictor. At the same time, they classify instructions depending on their latency so that the confidence threshold can be adapted to the potential gain of predicting a given instruction. Hence, operations with small gains are only predicted if the predictor's confidence is very high, whereas operations with potentially large gains are predicted even if the confidence is rather low. Interestingly, they found that loads are responsible for most of the latency in the critical path and hence predicting only loads represent a good filtering criteria. I implicitly use this filtering criteria because all my predictors only predict load values. Calder et al. further propose a *use*-bit that indicates if a predicted value has been consumed. If it has not, the comparison for validating the prediction can be omitted and no misprediction recovery is necessary even if the predicted value is incorrect. Finally, they suggest utilizing replacement and warm-up counters to minimize unnecessary replacements in their predictor and to delay predictions until the predictor has warmed up. I use a one-bit replacement counter in some of my predictors (Chapter 9). Instead of using warm-up counters, I set the confidence value to a low level after a replacement, which has the same effect, i.e., it inserts a delay before the confidence estimator starts allowing predictions.

Nakra, Gupta, and Soffa [NGS99a] present techniques to predict values based on global context, that is, the behavior of other instructions is used to guide the prediction process. For example, they propose a last value and a

stride predictor that make different predictions for the same instruction depending on the current execution path by storing different values and/or strides for each path. Furthermore, they suggest correlating predictions with recently completed instructions. Both techniques improve the predictor performance somewhat. I believe their predictors are good candidates for incorporating state reduction techniques similar to those discussed in Chapter 9, which should yield more cost effective implementations.

In their next paper Nakra, Gupta, and Soffa [NGS99b] investigate value prediction in connection with VLIW machines. Because such machines are statically scheduled and the code cannot easily be re-ordered at runtime, actual compensation code has to be included in the binary to support value prediction. In order to avoid the code increase and the performance impact of the compensation code, Nakra et al. describe a two-execution-unit system in which the second unit generates and executes the compensation code on-the-fly and concurrently with the regular program execution on the first execution engine.

Srinivasan and Lebeck [SrLe98] show that in some programs over sixty percent of the executed load instructions produce a value that is already needed in the next cycle. They further found that up to thirty-six percent of the loads miss in the L1 data-cache but have a latency demand that is lower than the L2 cache's access time. These percentages, which are likely to grow as the issue width increases and the speed-gap between CPUs and memory widens, show the growing need for a mechanism to reduce the load latency. Load value predictors are able to provide single and even zero cycle load latencies.

Morancho et al. [MLO98] propose separating the confidence estimator from the predictor so that only the confidence estimator has to be large enough to handle "all" load instructions, whereas the predictor itself can be designed smaller because it only has to hold the predictable loads. Unfortunately, such a scheme is not possible in my predictors because it is neces-

sary to also feed the unpredictable loads to the predictor so that predictability patterns can be established, which are essential for the *SAg* confidence estimator.

10.4.1 Dependence Prediction

In another paper [LiSh97], Lipasti and Shen add dependence prediction to their predictor and switch to predicting source operand values rather than instruction results to decouple dependence detection from value-speculative instruction dispatch. They found their approach to be particularly effective in wide and deeply pipelined machines.

Reinman and Calder [ReCa98] also examine dependence prediction and conclude that, due to the small hardware requirement, dependence prediction should be added to new processors first even though value prediction provides a larger performance improvement. Furthermore, they found both address prediction and memory renaming to be inferior to dependence and value prediction.

In another paper [RCT+98], Reinman et al. propose a software-guided approach for identifying dependencies between store and load instructions and devise an architecture to communicate dependency information to the hardware. Like other profile-based approaches, their approach requires changes to the instruction set architecture.

10.4.2 Confidence Estimation

Jacobsen et al. [JRS96], Tyson et al. [TLF97], and Grunwald et al. [GKMP98] introduce confidence estimation to the domain of branch prediction, dual-path, and multi-path execution in order to decide whether to make a prediction or whether to execute two or multiple program paths, respectively.

I adopt some of their metrics for load value prediction (see Section 3.4). While their goals are similar to mine, the approaches between branch confidence estimation and load value prediction differ. In particular, their confidence estimator (with two-bit saturating up/down counters) can be improved upon when applied to load value prediction (Chapter 5).

10.4.3 Branch Prediction

In the area of branch prediction, a significant amount of related work exists. Lee and Smith [LeSm84] propose keeping a *history* of recent branch directions for every conditional branch and systematically analyze the predictability of every possible pattern.

Yeh and Patt [YePa92, YePa93] and Pan, So, and Rahmeh [PSR92] describe sets of two-level branch predictors and invent a taxonomy to distinguish between them. I adopt their naming convention and one of their designs (the *SAg* predictor) for use as a confidence estimator in load value predictors.

Sprangle et al. [SCAP97] describe a technique called *agree prediction*, which reduces the chance that items mapped to the same predictor slot will interfere negatively. They achieve this by recording whether the previous branch predictions were correct or not instead of whether the branches were taken or not. I use a similar approach in the *SAg* confidence estimator by recording whether load values are predictable or not.

Chapter 11

Summary and Conclusions

One of the largest performance-bottlenecks in current microprocessors is the continuously growing load latency. Load value predictors can alleviate this problem by allowing the CPU to speculatively continue processing with a predicted load value while the memory access is still in progress. The goal of this dissertation is to improve the effectiveness of transparent, context-based load value predictors.

Most load value predictors contain confidence estimators to suppress uncertain predictions, which is essential for good performance. Analyzing the behavior of an existing confidence estimator revealed a deficiency that prevents it from handling sequences of alternating predictability effectively. Such sequences represent an important subset of the predictable load value sequences. To eliminate this weakness, I designed a different confidence estimator that is somewhat larger but yields higher speedups with most load value predictors.

In order to be effective in superscalar CPUs, load value predictors have to support multiple accesses per cycle. Splitting a predictor into several banks can provide the needed support for multiple simultaneous predictor accesses. My performance results show that a simple interleaved banking scheme can deliver the same speedup as a predictor that is able to handle an unlimited number of accesses per cycle.

An investigation of the utilization of load value predictors revealed that most of the predictor hardware is hardly ever or never used while a small part is used extremely frequently. To improve the utilization, I designed a predic-

tor that allocates more hardware to the frequently executed load instructions and is therefore able to correctly predict a larger number of loads, which improves the predictor performance considerably.

Furthermore, several sensitivity studies illustrate the range of parameters that result in good predictor performance. For instance, three to four-bit saturating counters (and ten-bit histories) seem to be necessary for good confidence estimator performance. Load value predictors should be at least 512 lines tall for the SPECint95 workload. While the performance delivered by a load value predictor varies greatly from program to program, the predictor configuration that yields the highest average performance over the SPECint95 benchmark suite also yields close to optimal performance for most of the individual programs, implying that a load value predictor's parameters do not have to be adapted to individual programs to yield a good average performance.

A study of the expressiveness of non-speedup-based metrics revealed that all of the investigated metrics appear to be misleading in some cases, indicating that genuine speedup measurements are probably required for performance evaluation purposes.

Furthermore, this dissertation presents performance numbers for a large number of load value predictors that are all evaluated in the same environment (i.e., the same simulator, the same benchmark programs, etc.), making it possible to truly compare the performance of the studied predictors.

An analysis of many hybrid predictor combinations shows that hybrids are able to deliver substantially more speedup than even the best single-component predictor because different components contribute independently to the overall performance. Studying the component's speedup contributions revealed that the register predictor with its poor individual performance represents a valuable addition to all other studied components. Conversely, combining well-performing predictors frequently does not result in an effective hybrid. In fact, some predictor combinations underperform a similar but smaller

hybrid due to negative interference.

The hybridization analysis allowed me to design a predictor using components that complement each other well. Other hybrids were found to contain components that do not ideally complement one another because they predict highly overlapping sets of load instructions.

Of all the studied predictors, the Reg+St2d+L4V predictor performs best with re-fetch as well as when averaging re-fetch and re-execute speedups. Since this predictor is rather large, I investigated techniques to reduce its size. I discovered that hybrids often store the same information in multiple components. Eliminating the redundant information can reduce the size of a component in a hybrid by more than a factor of two without compromising the performance. Another effective size-reduction method I developed is storing load values in a compressed format. A simple compression technique can reduce the predictor size by fifty percent while essentially maintaining the predictor's full performance.

Based on the well-performing Reg+St2d+L4V hybrid, I designed a very effective *coalesced-hybrid* load value predictor that requires only a small amount of state because it incorporates several of the developed state-reduction approaches. Cycle-accurate pipeline-level simulations of a four-way superscalar out-of-order CPU with many different predictors from the literature show that the coalesced-hybrid outperforms even the best predictors by almost twenty percent over a wide range of predictor sizes. In the smallest configuration I investigated, which requires fifteen kilobytes of state, the coalesced-hybrid yields a speedup with both a re-fetch and a re-execute misprediction recovery mechanism that surpasses the speedup of other predictors, including predictors that are over five times as large.

With fifteen kilobytes of state, the coalesced-hybrid's harmonic-mean re-fetch speedup over the eight SPECint95 programs is twelve percent and the re-execute speedup is fifteen percent. These performance improvements are obtained with a transparent load value predictor that can even be added to

existing CPU families because no change in the instruction set architecture is necessary. Furthermore, these speedups are obtained on programs that were not compiled with load value prediction in mind. In future work I will study compiler optimizations to further improve the performance of load value predictors.

Only value predictors are hybridized in this thesis. Since the number of transistors per chip continuously increases, more real-estate will soon be available for larger and more complex load value predictors. Hence, it may be worthwhile studying hybrid confidence estimators as well.

If the load latency continues to grow, and there is currently no indication that it will not, load value prediction will become more and more important. At some point it may even become necessary to predict the load latency so that more aggressive (but slower) load value predictors can be used to predict the long-latency load instructions. Thus, hierarchies of load value predictors with multiple levels may one day be commonplace.

Bibliography

[BJR+99] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz, U. Weiser. “Correlated Load-Address Predictors”. *26th International Symposium on Computer Architecture*. May 1999.

[BuZo98a] M. Burtscher, B. G. Zorn. “Profile-Supported Confidence Estimation for Load-Value-Prediction”. *PACT'98 Workshop on Profile and Feedback-Directed Compilation*. Paris, France. October 1998.

[BuZo98b] M. Burtscher, B. G. Zorn. *Load Value Prediction Using Prediction Outcome Histories*. Technical Report CU-CS-873-98, University of Colorado at Boulder. October 1998.

[BuZo99a] M. Burtscher, B. G. Zorn. “Prediction Outcome History-based Confidence Estimation for Load Value Prediction”. *Journal of Instruction-Level Parallelism*, Vol. 1. May 1999.

[BuZo99b] M. Burtscher, B. G. Zorn. “Exploring Last *n* Value Prediction”. *1999 International Conference on Parallel Architectures and Compilation Techniques*. October 1999.

[BuZo00] M. Burtscher, B. G. Zorn. *Coalescing and Hybridizing Load Value Predictors*. Technical Report CU-CS-903-00, University of Colorado at Boulder. March 2000.

[CaFe99] B. Calder, P. Feller, A. Eustace. “Value Profiling and Optimization”. *Journal of Instruction-Level Parallelism*, Vol. 1. March 1999.

[CFE97] B. Calder, P. Feller, A. Eustace. “Value Profiling”. *30th Annual ACM/IEEE International Symposium on Microarchitecture*. December 1997.

[CRT99] B. Calder, G. Reinmann, D. M. Tullsen. “Selective Value Prediction”. *26th International Symposium on Computer Architecture*. May 1999.

[DEC92] Digital Equipment Corporation. *Alpha Architecture Handbook*. 1992.

[Edm+95] J. H. Edmondson et al. “Superscalar Instruction Execution in the 21164 Alpha Microprocessor”. *IEEE Micro*, 15(2). April 1995.

[EuSr94] A. Eustace, A. Srivastava. *ATOM: A Flexible Interface for Building High Performance Program Analysis Tools*. WRL Technical Note TN-44, Digital Western Research Laboratory, Palo Alto. July 1994.

[FJLC98] C. Y. Fu, M. D. Jennings, S. Y. Larin, T. M. Conte. “Value Speculation Scheduling for High Performance Processors”. *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*. October 1998.

[Gab96] F. Gabbay. *Speculative Execution Based on Value Prediction*. EE Department Technical Report #1080, Technion - Israel Institute of Technology. November 1996.

[GaMe97] F. Gabbay, A. Mendelson. “Can Program Profiling Support Value Prediction?”. *30th Annual ACM/IEEE International Symposium on Microarchitecture*. December 1997.

[GaMe98] F. Gabbay, A. Mendelson. “The Effect of Instruction Fetch Bandwidth on Value Prediction”. *25th International Symposium on Computer Architecture*. June 1998.

[GKMP98] D. Grunwald, A. Klauser, S. Manne, A. Pleszkun. “Confidence Estimation for Speculation Control”. *25th International Symposium on Computer Architecture*. June 1998.

[GoGo98] J. Gonzalez, A. Gonzalez. “The Potential of Data Value Speculation to Boost ILP”. In *12th International Conference on Supercomputing*. 1998.

[GoGo99] J. Gonzalez, A. Gonzalez. “Control-Flow Speculation through Value Prediction for Superscalar Processors”. *1999 International Conference on Parallel Architectures and Compilation Techniques*. October 1999.

[Half95] T. R. Halfhill. “Intel’s P6”. *BYTE*, 20(4):42-58. April 1995.

[JRS96] E. Jacobsen, E. Rotenberg, J. Smith. “Assigning Confidence to Conditional Branch Predictions”. *29th International Symposium on Microarchitecture*. December 1996.

[John91] M. Johnson. *Superscalar Microprocessor Design*. Prentice Hall. 1991. ISBN 0-13-875634-1.

[Jou93] N. P. Jouppi. “Cache write policies and performance”. *Computer Architecture News, Proceedings of ISCA '20*, 191-201. May, 1993.

[KaEm91] D. R. Kaeli, P. Emma. “Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns”. *18th Annual International Symposium on Computer Architecture*, 19(3):34-42. May 1991.

[KMW98] R. E. Kessler, E. J. McLellan, D. A. Webb. “The Alpha 21264 Microprocessor Architecture”. *1998 International Conference on Computer Design*. October 1998.

[LCB+98] D. C. Lee, P. J. Crowley, J. J. Baer, T. E. Anderson, B. N. Bershad. "Execution Characteristics of Desktop Applications on Windows NT". *25th International Symposium on Computer Architecture*. June 1998.

[LCM97] C. Lee, I. Chen, T. Mudge. "The bi-mode branch predictor". *30th Annual ACM/IEEE International Symposium on Microarchitecture*, 4-13. December 1997

[LeSm84] J. K. F. Lee, A. J. Smith. "Branch Prediction Strategies and Branch Target Buffer Design". *IEEE Computer* 17(1):6-22. January 1984.

[LiSh96] M. H. Lipasti, J. P. Shen. "Exceeding the Dataflow Limit via Value Prediction". *29th International Symposium on Microarchitecture*. December 1996.

[LiSh97] M. H. Lipasti, J. P. Shen. "The Performance Potential of Value and Dependence Prediction". In *EUROPAR-97*. August 1997.

[LWS96] M. H. Lipasti, C. B. Wilkerson, J. P. Shen. "Value Locality and Load Value Prediction". *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, 138-147. October 1996.

[McF93] S. McFarling. *Combining Branch Predictors*. TN 36, DEC-WRL. June 1993.

[MLO98] E. Morancho, J. M. Llaberia, A. Olive. "Split Last-Address Predictor". *1998 International Conference on Parallel Architectures and Compilation Techniques*. October 1998.

[NGS99a] T. Nakra, R. Gupta, M. L. Soffa. "Global Context-based Value Prediction". *Fifth International Symposium on High Performance Computer Architecture*. January 1999.

[NGS99b] T. Nakra, R. Gupta, M. L. Soffa. "Value Prediction in VLIW Machines". *26th International Symposium on Computer Architecture*. May 1999.

[Pai96] A. Paithankar. *AINT: A Tool for Simulation of Shared-Memory Multiprocessors*. Master's Thesis, University of Colorado at Boulder. 1996.

[PSR92] S. T. Pan, K. So, J. T. Rahmeh. "Improving the Accuracy of Dynamic Branch Prediction using Branch Correlation". *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 76-84. October 1992.

[PeSm93] C. H. Perleberg, A. J. Smith. "Branch Target Buffer Design and Optimization". *IEEE Transactions on Computers*, 42(4):396-412. April 1993.

[RCT+98] G. Reinman, B. Calder, D. Tullsen, G. Tyson, T. Austin. *Profile Guided Load Marking for Memory Renaming*. Technical Report CS-98-593, University of California San Diego. July 1998.

[ReCa98] G. Reinman, B. Calder. "Predictive Techniques for Aggressive Load Speculation". *31st Annual ACM/IEEE International Symposium on Microarchitecture*. December 1998.

[RFKS98] B. Rychlik, J. Faistl, B. Krug, J. P. Shen. "Efficacy and Performance Impact of Value Prediction". *Proceedings of the 1998 International Conference on Parallel Architectures and Compiler Technology (PACT '98)*. October 1998.

[SaSm97a] Y. Sazeides, J. E. Smith. "The Predictability of Data Values". *30th Annual ACM/IEEE International Symposium on Microarchitecture*. December 1997.

[SaSm97b] Y. Sazeides, J. E. Smith. *Implementations of Context Based Value Predictors*. Technical Report ECE-97-8, University of Wisconsin-Madison. December 1997.

[SaSm98] Y. Sazeides, J. E. Smith. "Modeling Program Predictability". *25th International Symposium on Computer Architecture*. June 1998.

[SCAP97] E. Sprangle, R. Chappell, M. Alsup, Y. Patt. "The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference". *24th Annual International Symposium of Computer Architecture*, 284-291. 1997.

[SLM95] S. Sechrest, C. C. Lee, T. Mudge. "The Role of Adaptivity in Two-level Adaptive Branch Prediction". *28th International Symposium on Microarchitecture*. 1995.

[SmSo95] J. E. Smith, G. S. Sohi. "The Microarchitecture of Superscalar Processors". *Proceedings of the IEEE*. 1995.

[SoSo98] A. Sodani, G. S. Sohi. "Understanding the Differences Between Value Prediction and Instruction Reuse". *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, 205-215. 1998.

[SrLe98] S. T. Srinivasan, A. R. Lebeck. "Load Latency Tolerance in Dynamically Scheduled Processors". *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, 148-159. 1998.

[SPEC95] *SPEC CPU'95*. August 1995.

[SrEu94] A. Srivastava, A. Eustace. “ATOM: A System for Building Customized Program Analysis Tools”. *ACM SIGPLAN ’94 Conference on Programming Language Design and Implementation*. ACM SIGPLAN 29(6):196-205. June 1994.

[SrWa93] A. Srivastava, D. Wall. “A Practical System for Inter-module Code Optimization at Linktime”. *Journal of Programming Languages* 1(1). March 1993.

[TLF97] G. Tyson, K. Lick, M. Farrens. *Limited Dual Path Execution*. Technical Report CSE-TR-346-97, University of Michigan. 1997.

[TuSe99] D. Tullsen, J. Seng. “Storageless Value Prediction Using Prior Register Values”. *26th International Symposium on Computer Architecture*. May 1999.

[WaFr97] K. Wang, M. Franklin. “Highly Accurate Data Value Prediction using Hybrid Predictors”. *30th Annual ACM/IEEE International Symposium on Microarchitecture*. December 1997.

[Yeag96] K. C. Yeager. “The MIPS R10000 Superscalar Microprocessor”. *IEEE Micro*, 28-40. April 1996.

[YePa92] T. Y. Yeh, Y. N. Patt. “Alternative Implementations of Two-level Adaptive Branch Prediction*”. 19th Annual International Symposium of Computer Architecture*, 124-134. May 1992.

[YePa93] T. Y. Yeh, Y. N. Patt. “A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History*”. 20th Annual International Symposium of Computer Architecture*, 257-266. May 1993.

[You94] J. L. Young. *Insider’s Guide to PowerPC Computing*. Que Corporation. 1994. ISBN 1-56529-625-7.

Appendix

Appendix A

Theorem: If the runtime of the individual programs of a benchmark suite is normalized for CPU_{Base} (i.e., the CPU without any load value predictor), the combined speedup evaluates to the *harmonic mean* of the individual speedups. If the normalization is done for CPU_{LVP} (i.e., the CPU with the load value predictor), the combined speedup evaluates to the *arithmetic mean* of the individual speedups.

Proof: Suppose there are n programs with the runtimes $RT_{Base,i}$ and $RT_{LVP,i}$ for $i = 1, 2, ..., n$ on CPU_{Base} and CPU_{LVP}, respectively.

By definition, the individual speedups are

$$s_i = \frac{RT_{Base,i}}{RT_{LVP,i}}$$

When normalizing the baseline runtimes to be one, we obtain

$$RT^*_{Base,i} = \frac{RT_{Base,i}}{RT_{Base,i}} = 1$$

$$RT^*_{LVP,i} = \frac{RT_{LVP,i}}{RT_{Base,i}}$$

where the starred values represent the normalized values. We then find the overall speedup S (i.e., the total runtime on the baseline processor over the total runtime on the processor with the load value pre-

dictor) to be the harmonic mean of the individual speedups.

$$S = \frac{\sum_i RT^*_{Base,i}}{\sum_i RT^*_{LVP,i}} = \frac{n}{\sum_i RT^*_{LVP,i}} = \frac{n}{\sum_i \frac{RT_{LVP,i}}{RT_{Base,i}}} = \frac{n}{\sum_i \frac{1}{s_i}} = HM(s_1, s_2, ..., s_n)$$

Similarly, when normalizing the CPU_{LVP} runtimes to be one, we get

$$RT^*_{LVP,i} = \frac{RT_{LVP,i}}{RT_{LVP,i}} = 1$$

$$RT^*_{Base,i} = \frac{RT_{Base,i}}{RT_{LVP,i}}$$

and now find the overall speedup *S* to be the arithmetic mean of the individual speedups.

$$S = \frac{\sum_i RT^*_{Base,i}}{\sum_i RT^*_{LVP,i}} = \frac{\sum_i RT^*_{Base,i}}{n} = \frac{\sum_i \frac{RT_{Base,i}}{RT_{LVP,i}}}{n} = \frac{\sum_i s_i}{n} = AM(s_1, s_2, ..., s_n)$$

♦

Appendix B

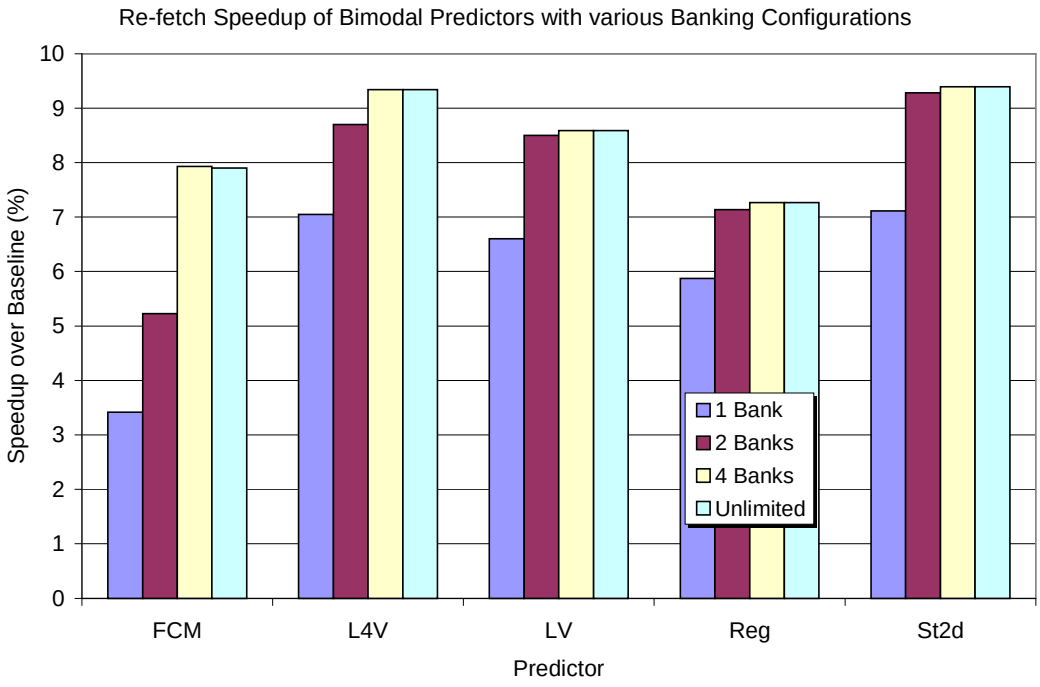

Figure 12.1: Re-fetch speedup of differently banked *bimodal* predictors.

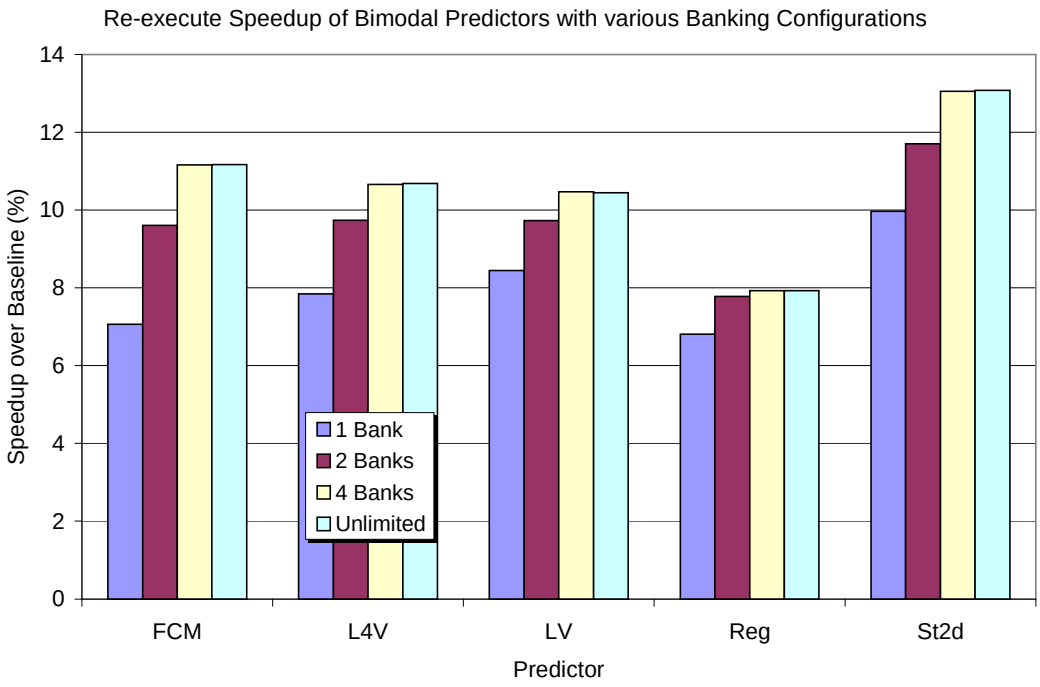

Figure 12.2: Re-execute speedup of differently banked *bimodal* predictors.

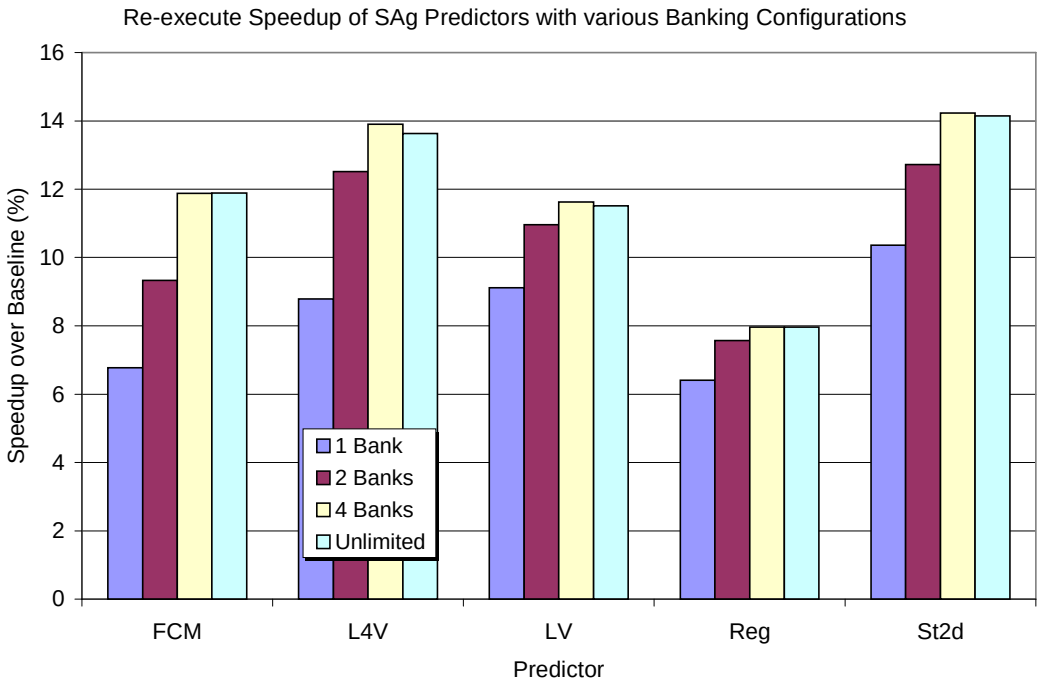

Figure 12.3: Re-execute speedup of differently banked *SAg* predictors.

number of banks	recovery mechanism	confidence estimator	load value predictor	load instrs per cycle	accesses per cycle	references per cycle	updates per cycle	% dropped predictions	% dropped updates
1	re-fetch	bimodal	FCM	0.410	0.794	0.403	0.391	49.8	20.5
1	re-fetch	bimodal	L4V	0.426	0.779	0.395	0.384	50.1	25.1
1	re-fetch	bimodal	LV	0.424	0.780	0.395	0.384	50.0	24.5
1	re-fetch	bimodal	Reg	0.421	0.774	0.391	0.383	50.8	23.7
1	re-fetch	bimodal	St2d	0.425	0.790	0.403	0.387	50.3	24.5
1	re-fetch	SAg	FCM	0.412	0.790	0.401	0.389	49.8	20.5
1	re-fetch	SAg	L4V	0.424	0.791	0.402	0.389	51.1	24.3
1	re-fetch	SAg	LV	0.424	0.791	0.403	0.388	50.6	24.2
1	re-fetch	SAg	Reg	0.420	0.774	0.391	0.383	50.5	23.7
1	re-fetch	SAg	St2d	0.425	0.800	0.408	0.392	51.1	24.1
1	re-execute	bimodal	FCM	0.425	0.780	0.390	0.390	50.0	22.7
1	re-execute	bimodal	L4V	0.429	0.781	0.395	0.386	50.0	24.9
1	re-execute	bimodal	LV	0.431	0.787	0.396	0.390	50.4	24.9
1	re-execute	bimodal	Reg	0.426	0.776	0.392	0.384	50.3	24.7
1	re-execute	bimodal	St2d	0.436	0.796	0.400	0.396	50.4	25.1
1	re-execute	SAg	FCM	0.423	0.780	0.388	0.391	50.3	22.1
1	re-execute	SAg	L4V	0.433	0.785	0.395	0.390	50.5	25.4
1	re-execute	SAg	LV	0.434	0.786	0.397	0.389	50.3	25.6
1	re-execute	SAg	Reg	0.424	0.776	0.391	0.384	50.8	24.1
1	re-execute	SAg	St2d	0.438	0.797	0.396	0.400	50.7	24.2
2	re-fetch	bimodal	FCM	0.418	0.977	0.501	0.476	19.3	1.7
2	re-fetch	bimodal	L4V	0.434	0.991	0.504	0.487	18.9	2.2
2	re-fetch	bimodal	LV	0.433	0.989	0.504	0.485	18.9	2.2
2	re-fetch	bimodal	Reg	0.428	0.980	0.500	0.479	19.0	2.2
2	re-fetch	bimodal	St2d	0.436	1.002	0.514	0.488	19.0	2.2
2	re-fetch	SAg	FCM	0.416	0.978	0.505	0.473	19.2	1.8
2	re-fetch	SAg	L4V	0.442	1.013	0.515	0.498	19.1	2.1
2	re-fetch	SAg	LV	0.437	1.010	0.517	0.493	19.0	2.2
2	re-fetch	SAg	Reg	0.428	0.980	0.500	0.480	19.0	2.1
2	re-fetch	SAg	St2d	0.438	1.016	0.522	0.494	19.0	2.2
2	re-execute	bimodal	FCM	0.441	0.987	0.492	0.495	19.1	1.9
2	re-execute	bimodal	L4V	0.437	0.997	0.506	0.491	19.0	2.1
2	re-execute	bimodal	LV	0.436	0.996	0.504	0.492	19.0	2.2
2	re-execute	bimodal	Reg	0.430	0.981	0.500	0.481	19.0	2.2
2	re-execute	bimodal	St2d	0.442	1.012	0.510	0.502	19.1	2.1
2	re-execute	SAg	FCM	0.439	0.987	0.495	0.493	19.1	1.9
2	re-execute	SAg	L4V	0.453	1.013	0.508	0.506	19.0	2.0
2	re-execute	SAg	LV	0.442	1.005	0.507	0.498	19.0	2.1
2	re-execute	SAg	Reg	0.429	0.980	0.499	0.481	19.0	2.2
2	re-execute	SAg	St2d	0.448	1.020	0.513	0.507	18.9	2.0
4	re-fetch	bimodal	FCM	0.433	1.100	0.603	0.497	0.0	0.017
4	re-fetch	bimodal	L4V	0.438	1.106	0.603	0.502	0.0	0.020
4	re-fetch	bimodal	LV	0.434	1.101	0.603	0.498	0.0	0.029
4	re-fetch	bimodal	Reg	0.428	1.092	0.600	0.492	0.0	0.013
4	re-fetch	bimodal	St2d	0.437	1.116	0.615	0.501	0.0	0.027
4	re-fetch	SAg	FCM	0.428	1.097	0.605	0.493	0.0	0.015
4	re-fetch	SAg	L4V	0.455	1.141	0.618	0.522	0.0	0.020
4	re-fetch	SAg	LV	0.440	1.130	0.619	0.511	0.0	0.039
4	re-fetch	SAg	Reg	0.429	1.093	0.600	0.493	0.0	0.013
4	re-fetch	SAg	St2d	0.440	1.137	0.626	0.511	0.0	0.030
4	re-execute	bimodal	FCM	0.451	1.108	0.595	0.513	0.0	0.032
4	re-execute	bimodal	L4V	0.442	1.112	0.604	0.508	0.0	0.041
4	re-execute	bimodal	LV	0.439	1.116	0.606	0.510	0.0	0.042
4	re-execute	bimodal	Reg	0.430	1.093	0.599	0.494	0.0	0.010
4	re-execute	bimodal	St2d	0.447	1.136	0.613	0.523	0.0	0.049
4	re-execute	SAg	FCM	0.459	1.116	0.597	0.519	0.0	0.027
4	re-execute	SAg	L4V	0.462	1.136	0.607	0.529	0.0	0.036
4	re-execute	SAg	LV	0.445	1.123	0.608	0.515	0.0	0.029
4	re-execute	SAg	Reg	0.431	1.095	0.599	0.496	0.0	0.014
4	re-execute	SAg	St2d	0.453	1.142	0.615	0.528	0.0	0.019

Table 12.1: Banking information.

Appendix C

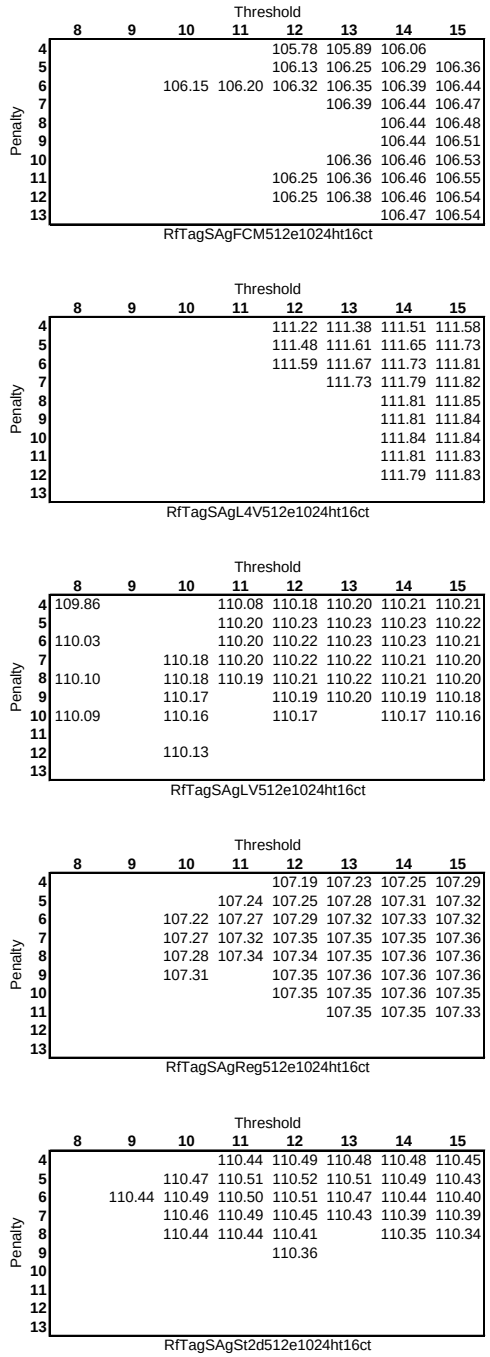

Figure 12.4: The re-fetch speedup maps for the five basic *SAg* predictors.

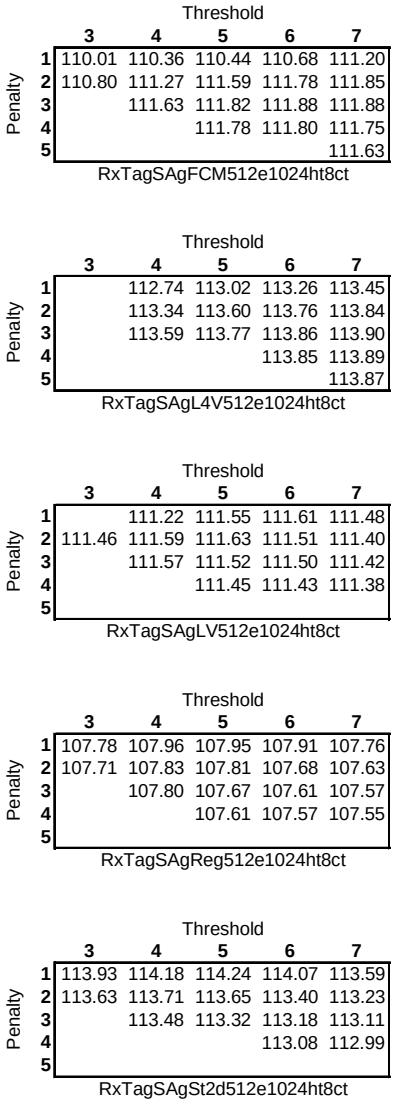

Figure 12.5: Re-execute speedup maps for the five basic *SAg* predictors.

Appendix D

Naming Conventions

Load value predictors consist of several components, including tags, confidence estimators, and value predictors. I chose predictor names that are abbreviations of the various parts that make up a load value predictor, as described below.

Tags: The lines in a load value predictor may optionally be tagged. If tags are present, they are usually shared among the value predictor and the confidence estimator. In this dissertation, the word *tag* in front of the predictor name indicates the presence of (partial) tags.

Confidence Estimators: Strictly speaking, the confidence estimator is also optional, but to date there exists no proposed predictor that performs well without one (see Section 5.1).

Several papers have been published that define taxonomies for branch predictors. Since confidence estimators are structurally identical to branch predictors, I will adhere to the established branch predictor nomenclature when describing confidence estimators (e.g., *SSg*, *SAg*, *bimodal*, etc.).

Yeh and Patt introduced a taxonomy for two-level (branch) predictors [YePa92, YePa93] that consist of a branch history register (BHR) and a pattern history table (PHT). They use three-letter combinations to describe the two components. By convention, the first two letters are uppercase and the third letter is lowercase.

The first letter characterizes the BHR. If all branches share a common BHR, a *G* is used to indicate *global*. If every branch has its own BHR, a *P* is used to indicate *per-address*. If sets of branches are mapped to individual

BHRs, an *S* is used to indicate *set*. Note that *G* and *P* represent the two extremes of the set case. *P* means all sets have a size of one and *G* means there is only one set that encompasses all the branches.

The second letter specifies whether the PHT is adaptive. *S* stands for *static*, indicating that the PHT entries are fixed. *A* stands for *adaptive*, meaning that the PHT entries can be modified dynamically.

The third (lowercase) letter is identical to the first letter, except it describes the PHT instead of the BHR. Hence, *g* means one global PHT, *s* means one PHT per set, and *p* means one PHT per address.

The most frequently used confidence estimators in this thesis are the *SAg* confidence estimator [YePa93] and the *bimodal* confidence estimator [McF93] (Chapter 5).

Value Predictors: I use abbreviations of the kind of information that the predictor retains as a predictor name. For instance, a predictor that retains the *last four values* is called an L4V predictor. Other examples are St2d for *stride 2-delta* predictor, Reg for *register* file predictor, FCM for *finite context method* predictor, etc.

Hybrid predictors, that is, predictors that contain multiple value predictor components, are named after their individual components, delimited by a plus sign (e.g., a hybrid between a *stride* and a *last value* predictor would be called an St+LV predictor).