A FRAMEWORK FOR PERFORMING PREDICTION IN VLIW ARCHITECTURES

by

Tarun Nakra

B.S. Computer Science
University of Roorkee, India, 1994

Submitted to the Graduate Faculty of
Arts and Sciences in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

University of Pittsburgh
2001

UNIVERSITY OF PITTSBURGH

FACULTY OF ARTS AND SCIENCES

This dissertation was presented

by

Tarun Nakra

It was defended on

June 11, 2001

and approved by

Dr. Mary Lou Soffa

Dr. Bruce R. Childers

Dr. Rami Melhem

Dr. Ray R. Hoare (EE)

Committee Chairperson

A FRAMEWORK FOR PERFORMING PREDICTION IN VLIW ARCHITECTURES

Tarun Nakra, Ph.D.

University of Pittsburgh, 2001

Recently, Very Long Instruction Word (VLIW) architectures have gained popularity with the advent of EPIC computing and several embedded processors adopting the VLIW computing model. These architectures do not involve run-time reordering of instructions and have lower hardware complexity than out-of-order processors. The performance of a VLIW processor is dependent on the capability of the compiler to statically detect and exploit instruction-level parallelism. Static scheduling of instructions allows reordering over a larger scope than the scheduling window of a dynamically schedulable processor. However, accurate run-time information is not available at compile-time, and the compiler traditionally preserves the conservative program dependencies while scheduling. Some of the program dependencies can be overcome by speculatively executing instructions. Speculation predicts the execution behavior of instructions thereby increasing the number of instructions executing in parallel. This dissertation describes a conceptual framework that enables speculative execution in VLIW processors using prediction techniques. The framework design aims to facilitate translation of a prediction-driven optimization into the VLIW compiler and architecture. The framework has been applied to develop profile-driven optimizations that predict attributes of instruction data, namely computed values and operand bit-widths. Value prediction predicts an instruction value enabling speculative execution of instructions dependent on the predicted value. Bit-width prediction predicts widths of instruction operands allowing instructions with narrow operands to be packed together on a functional unit. Within each optimization, the framework application integrates static analysis and run-time information in a synergistic way. Profiling is used to estimate run-time behavior during static scheduling, and the cost of profiling is reduced by statically identifying the useful predictions of a program. The predictions are performed using novel schemes and captured at run-time by hardware that is able to effectively estimate the dynamic predictability behavior of instructions. The predictions are exploited using compiler transformations that speculate aggressively and recovery schemes that reduce the overhead of misprediction. The efficacy of the optimizations implemented using the framework is evaluated through empirical evaluations on realistic VLIW machine models. The experimental results indicate the framework to be quite effective in addressing the challenges of fine-grain parallelism in VLIW architectures.

Acknowledgments

I would like to thank my committee: Mary Lou Soffa, Bruce Childers, Rami Melhem and Ray Hoare, for their comments that helped organize this dissertation. I am most indebted to my advisor, Mary Lou Soffa, whose advice, insight and encouragement played a key role in the development of this dissertation. I am grateful to Mary Lou for helping shape my graduate school career productively, and for her support during challenges I had to face during this period. I am also thankful to Bruce Childers for his advice and joint work that were contributing to this thesis. I am grateful to Rajiv Gupta for the useful conversations and inspiring ideas that helped define the direction of this work. I would also like to thank John Shen for his encouragement to pursue this research. Within the computer science department at Pittsburgh, I am thankful to Atif Memon and Clara Jaramillo for sharing an office as well as several memories that will last for a long time.

I am grateful to my parents in India for having supported me during the period of my graduate studies. My father provided a role model for me to realize the value of dedication and perseverance for seeking the goals I had set as part of this work. I am also thankful to my extended family in India and to my wife's family in Pittsburgh for their constant encouragement. Finally, I would like to thank my wife Christine for her love and support throughout my graduate education. Her friendship provided me the sunshine and joy that encouraged me to carry out my endeavors with a smile.

To the family I treasure - my parents, brother Mohit,
sister-in-law Meenakshi, and the love of my life Christine

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the last two decades, RISC architecture design has had a major impact on processor technology, leading to innovative designs of processors. One of the important subclasses of RISC processors is the class of the *very long instruction word* (VLIW) architectures [37]. A VLIW processor executes multiple operations in parallel using separate functional units. Instructions are fetched from the cache in the form of Very-Long Instruction Words containing multiple primitive operations, and an entire instruction is issued for parallel execution of its operations. The execution does not involve any run-time reordering of operations, as in conventional superscalar processors. Instead, the compiler performs static scheduling and generates code that has independent primitive operations grouped together in an instruction for parallel execution. Using the compiler reduces the complexity of the processor design, and the VLIW architecture consists of an execution engine with simple control logic. Hardware complexity is reduced by avoiding out-of-order execution because scheduling is done at compile-time. A VLIW processor is scalable to a wider fetch window as well as larger number of functional units. VLIW architectures have recently gained popularity with the advent of EPIC computing [66] and several embedded processors adopting the simplistic static scheduling model.

The performance of VLIW processors depends on the capability of the compiler to statically identify the operations to group together and fill the available operation slots in an instruction word. The compiler performs this grouping by uncovering instruction-level parallelism (ILP) from the program. Executing operations in parallel exploits the available functional units and improves performance by reducing the instruction schedule length. Many performance-improving code optimizations have been proposed and developed for VLIW compilers to help exploit the functional units [22, 25, 13]. Some of these optimizations are applied at the source level, or at the high-level intermediate code preserving the control structure and sequencing in the source-code representation. Also a number of code optimizations are machine-dependent, and operate on a low-level representation of code.

One of the classes of optimizations in the latter category is the class of *prediction-based optimizations*. These optimizations increase parallelism by predicting the execution behavior of instructions prior to their actual execution. These predictions allow speculative execution of instructions and the speculation increases parallel execution of the instructions.

Programs, in general, exhibit execution characteristics that are frequently repeated. Prediction-based optimizations attempt to utilize this predictable behavior of program execution for performance enhancements. This predictable behavior arises due to significant amounts of control locality in programs and can be explained by the 80-20 rule. Although this rule originally referred to programs spending about 80% of execution time in 20% of code, it is also valid in the domain of executed paths. Within this domain, programs are found to spend the majority of their time within a subset of executed paths, namely hot paths. These hot paths generate predictable values since programs repeat similar computations along the same execution path. This predictable behavior is apparent in several aspects of program execution. For example, studies have observed that 40-50% of values computed by an instruction at run-time are exactly the same as the values computed in previous instances of the instruction's execution [48]. Also, memory reference patterns are frequently repeated and memory dependencies can be predicted accurately by chaining the operations aliasing the same location [15].

Superscalar processors have been quite successful in exploiting predictions at run-time for performance enhancements. These processors capture predictable behavior by storing the history of previous computations in hardware to use to predict the program's future behavior. For example, commercial superscalar processors incorporate prediction in the form of branch predictors (Alpha 21264, PowerPC 620) and trace caches (Pentium 4). Performing predictions at run-time allows these processors to use accurate dependency information and to dynamically control the aggressiveness of prediction. Lately, due to the emergence of several commercial VLIW processors, interest in developing compiler-based prediction optimizations has been increasing. In the compiler domain, such optimizations can be useful to speculatively hoist code during static scheduling. The major focus of the research in this dissertation is the design of prediction-based optimizations for VLIW processors.

There are several attributes of programs that, if correctly predicted, can reduce execution time on a parallel processor. Execution of statements in a program is constrained by a number of dependencies. Several of these dependencies, such as control, memory and true data dependencies, occur because of the program structure. These dependencies can be overcome by predicting them accurately, allowing the dependent instruction to speculatively execute before the dependence gets resolved. Figure 1.1(a) shows an example of such a situation. In this figure, the instructions shown are constrained to execute sequentially due to true data dependencies between them; r_3 depends on r_2 which depends on r_1. Predicting the first of these dependencies, as shown in Figure 1.1(b), allows the last two instructions to execute in parallel with the first instruction, since the latency of the load of r_1 is higher than the add instructions. The prediction is confirmed using a verification stage, which may incur an additional cycle. If the prediction is correct, the prediction improves the schedule. In the case of mispredictions, speculated instructions that were incorrectly executed need to be re-executed with verified values.

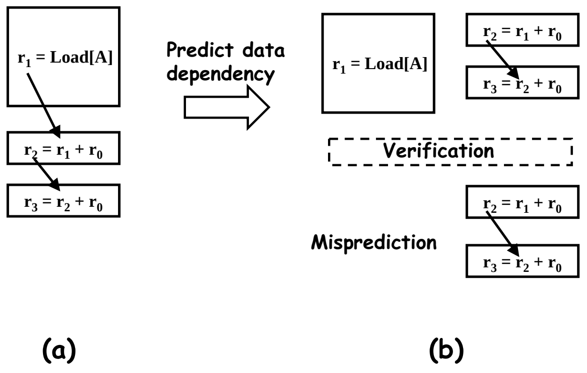


Figure 1.1: Prediction of Program Dependence

Program execution is also constrained by dependencies introduced by the processor model due to limitations of resources. Among these dependencies are artificial (anti and output) dependencies and resource dependencies. Figure 1.2(a) demonstrates a case when resource limitations serialize program execution. In this figure, assume the underlying machine model consists of a single 32-bit ALU. This restriction prevents the two add instructions from executing in parallel even though the program itself does not serialize those instructions. If both of these instructions operate on 16-bit data, these instructions can be packed together on the same ALU. Figure 1.2(b) shows the packed instructions for such a case. The verification process depends on the architecture implementation, and in the example, verification is performed in parallel with the speculation. Mispredictions impact the benefits of speculation by incurring re-execution of those packed instructions that require more than 16-bits for run-time computation. In both examples, if the predictable behavior of a program's execution is well captured by making accurate predictions, speculative execution can improve program performance.

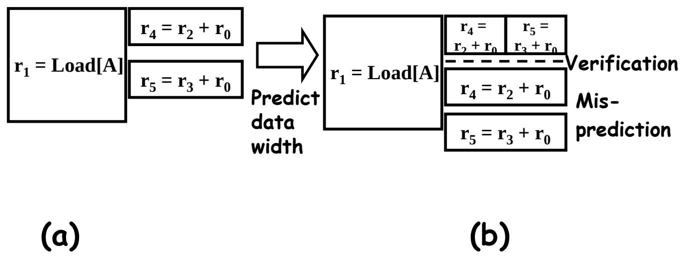


Figure 1.2: Prediction of Resource Dependence

1.1 Challenges of prediction in VLIW architectures

The advantage of performing static scheduling in a VLIW architecture is that the compiler can exploit opportunities for parallelism by using a global instruction window that is beyond basic blocks. However incorporating prediction-driven optimizations in a VLIW architecture through a compiler involves a number of challenges.

Due to the static scheduling model, the dependency information available at compile-time for scheduling is typically very conservative. For example, memory aliasing cannot be completely resolved until run-time and restricts scheduling of load instructions before potentially conflicting stores. In order to effectively incorporate prediction, a VLIW compiler needs accurate run-time information to guide the prediction process. Profiling has been used in the context of VLIW architectures to provide feedback to the compiler for speculating instructions [19, 28]. However, profiling at the instruction-level is very expensive. As an example, the *pixie* profiling system incurs 100-200% execution overhead if each basic block is profiled [39]. Thus, if prediction is implemented at the instruction level, the overhead of the profiler can become a serious bottleneck in designing an optimization. One of the challenges of prediction in the static domain is to keep the profiling overhead within reasonable limits.

The static scheduling model of the VLIW architecture enforces the condition that the instructions to predict be selected at compile-time because the instruction schedule must be set prior to execution. Hence, if an instruction is selected for prediction at compile-time, all instances of that instruction at run-time are predicted at run-time. The instructions are statically selected based on their predictability behavior within the profile data. In order for a large fraction of instructions to meet a threshold of predictability from the profile data, accurate prediction schemes need to be designed. These prediction schemes would not only be used by the profiler to gauge instruction predictability, but would also be incorporated within the predictor for run-time predictions. Designing effective prediction schemes is a significant issue for prediction-based optimizations on a VLIW processor.

Also performing accurate predictions is not sufficient since the predictions, although accurate, may not improve performance in some cases. It is possible for a prediction to be accurate but fail to trigger any speculation because of the lack of any instructions in the program that can benefit from the prediction. Thus, only the predictions useful for improving performance should be selected by the compiler. Since predictions are selected at compile-time, static analysis of the program should be used to identify these profitable predictions.

It has been observed that many instructions are easily predictable in a limited number of *phases* of their execution lifetime [68]. During other phases, these instructions are either predictable using a more complex scheme or they are not predictable at all. The instructions predictable using complex schemes can be addressed by designing predictors that adapt to different prediction schemes during phase changes at run-time. In the VLIW domain, because of the simple execution engine, design of such predictors needs to ensure that processor complexity is not severely impacted. If complex hardware is needed for making the predictions, the advantage of a simplistic VLIW architectural model is lost.

In the case when an instruction frequently shows transient behavior of unpredictability during its execution, even sophisticated predictors fall short of predicting very accurately. In such cases, mispredictions cannot be avoided because the static scheduling VLIW model prevents predictions from being turned on and off at run-time based on their dynamic behavior. In order to handle such cases, misprediction recovery schemes need to be designed that minimize performance degradation during mispredictions. Designing efficient recovery schemes is also important because handling mispredictions is expensive in the VLIW domain due to the fixed scheduling model. If a misprediction is encountered, the in-order execution of a VLIW processor prevents the execution of subsequent instructions encountered before completing recovery. This recovery cannot be masked by keeping the processor busy with out-of-order instruction issue, as is the case with superscalar processors. Using only software compiler techniques, recovery can be performed by generating recovery blocks for the misprediction cases [28]. Software-based recovery has effects on code size and cache performance that may limit any benefits of speculation. Hence handling mispredictions is quite challenging when developing prediction-based optimizations in the VLIW domain.

Thus, the VLIW model of computation introduces several challenges for designing prediction-based optimizations including making accurate and useful predictions, modeling phase changes in predictions and handling mispredictions efficiently and effectively. These challenges are applicable to any VLIW optimization utilizing prediction. The prediction techniques that have been proposed for VLIW architectures have been designed in isolation without providing a general methodology for prediction. Since there are a number of characteristics of program execution that can be predicted, a unified framework providing conceptual tools that every prediction-based optimization would need, would be beneficial. Such a general framework would enable a novel prediction-based optimization to be easily implemented within a VLIW architecture and compiler. The major goal in this research is the development of such a general framework.

1.2 Overview of research

This dissertation explores approaches for performing prediction in VLIW architectures. This research first develops a general methodology that provides a conceptual view of the components within a prediction-driven optimization. This methodology is then applied to develop specific optimizations that address the challenges highlighted in the previous section. The development of the optimizations involve addressing the following goals related to predicting in VLIW architectures.

- Designing effective prediction schemes that are able to make accurate predictions on the underlying instruction attribute being predicted within the optimization.
- Identify the regions of a program where predictions would benefit performance, by developing static analysis to establish usefulness of predictions.
- Exploit the useful predictions through profile-driven compile-time speculation.
- Keep the cost of obtaining the profile information within reasonable limits.
- Design the predictor to make the predictions at run-time, and the predictor design should aim at low cost while maintaining the accuracy of the prediction schemes.
- Develop techniques to handle the mispredictions within the optimization efficiently.

1.2.1 Framework

The above goals are addressed by first developing a general framework for representing prediction in VLIW architectures at a high-level. This framework is instrumental in developing solutions for the above goals and facilitating the translation of a prediction-based optimization into a VLIW compiler and architecture. The framework consists of the following components:

- *Usefulness Analyzer*: Identifies the program points where prediction, if applied, would improve overall performance.
- *Selective Profiler*: Profiles the program instructions using the prediction schemes to test predictability.
- *Speculation Transformer*: Transforms code by speculatively hoisting instructions that display predictable behavior.
- *Predictor*: Handles the predictions at run-time.
- *Restorer*: Restores program state during mispredictions.

The framework integrates ideas from the domains of profiling, compilation and architecture design. Each domain has its own unique strengths and the integration of the domains offers strengths not present in a single domain. This synergistic approach was used to develop the prediction framework. The practical utility of the framework was tested by developing specific prediction-based optimizations. Before implementing the above components for any optimization, the predictability was estimated by designing prediction schemes for the optimizations. A brief overview of the optimizations developed for prediction in VLIW processors is given in the next section.

1.2.2 Prediction Optimizations

This dissertation utilizes the framework to develop prediction-driven optimizations including value prediction and bit-width prediction. For each optimization, the framework components were developed and the goals mentioned in Section 1.2 were addressed during their implementation. Several of the ideas developed as part of this implementation are general enough to extend their scope of applicability across multiple architectural platforms.

1.2.2.1 Value Prediction

Value prediction predicts the values computed by an instruction. The advantage of performing value prediction is that instructions dependent on the predicted instruction can be speculatively executed using the predicted value. In this way, delays caused by true data dependencies existing between an instruction computing a value and the dependent instructions using that value can be avoided. The true data dependencies are a serious bottleneck in restricting the capability of a processor to exploit parallelism and occur more frequently than control dependencies. Figure 1.1 gave an example of value prediction improving instruction schedule upon correct prediction.

The framework was used to develop an optimization for predicting values computed by load instructions. Since loads have long latencies, predicting their values enables dependent instructions to overcome these latencies. Initially, the predictability of load values was analyzed by applying various techniques for prediction. Previously proposed value prediction techniques used the local context of an instruction to predict future values. One of the contributions to this thesis was to develop novel value prediction schemes that used a more global context associated with an instruction for prediction [56]. The types of global context used included path information as well as values produced by recent execution instances of the instruction. These techniques along with previously proposed prediction techniques were used to generate value profiles. Among the framework components, the usefulness analyzer was designed to use dataflow analysis to compute the change in critical path of a region due to a prediction. Predictions that did not improve the critical path were not instrumented by the selective profiler. Usefulness reduced the cost of value profiling significantly. The profile information generated by the selective profiler was exploited by the speculation transformer to transform the code enabling *value speculation* [28] of instructions that used predicted values. Instructions to predict and validate values were added to the instruction stream, exposing more parallelism to the scheduler. Hence, the compiler scheduled the instructions to predict within a program as well as those instructions that can be speculated using the predictions.

The value prediction techniques used in the framework predicted the next value of an instruction using *run-time history* of predictions. Run-time prediction history is the past predictability behavior of an instruction that can be used to make the next prediction for that instruction. Several optimizations use run-time history to make predictions, such as branch prediction using previous outcomes of branches to predict the next outcome. The run-time predictor for value prediction was developed in hardware and included prediction tables that store the history of predictions for instructions. A majority of program predictions were found to be accurately predicted only if the prediction technique varied dynamically using a hybrid prediction scheme. The value predictor was implemented as a *segmented predictor* in which different partitions of the predictor stored different amounts of run-time history. This hardware partitioning was implemented using feedback information about program predictions that required hybrid prediction as opposed to the ones predictable using just a single scheme. Incorporating a segmented predictor reduced the cost of implementing hybrid prediction significantly. Compiler analysis was used to configure the predictor for a program by annotating the predicted instructions with table entries that contained the corresponding predicted values. Each prediction was assigned to one of the partitions of the predictor, based on the most profitable prediction schemes for that prediction, as indicated by the profiler.

Misprediction recovery was addressed by developing the restorer in both the static and dynamic domains and comparing their performances to choose the more efficient alternative. In the static domain, recovery blocks were inserted in the code that were executed to recover from a misprediction. In the dynamic domain, a *compensation code engine* architecture was designed to generate and execute recovery code at run-time [57]. This engine allowed the original VLIW code to execute without stalling when recovery of a misprediction took place. The engine also avoids any increase in static code size due to prediction. Value prediction was analyzed on realistic VLIW machine models with significant performance speedups for the dynamic recovery case.

1.2.2.2 Bit-width prediction

The framework was also used for another prediction-driven optimization, bit-width prediction, that predicts widths of instruction operands, allowing instructions with narrow operands to be packed together on a functional unit. In contrast to value prediction which is beneficial for extracting higher degrees of instruction-level parallelism, bit-width prediction is useful when the underlying machine model cannot handle the available ILP in the program. Lately, multimedia applications have become an integral part of workload characterizations of microprocessors and these applications typically operate on small operand widths. Also, as processor word widths shift to 32 and 64 bits, the general purpose applications typically do not require all of the bits available for processing. Bit-width prediction attempts to pack operations with narrow operands together to improve resource utilization.

A prediction scheme for bit-width prediction was developed for packing instructions with narrow operands on the same ALU. *Bit-width profiling* was developed as a component in the framework to compute how often the width of an operand was found to be within a byte or half-word. The profile data was used by the speculation transformer to identify instructions whose operands were narrow enough to be packed on a functional unit together with other similar instructions. Static analysis was used by the usefulness analyzer to determine the regions within the program that did not have enough parallelism and bit-width profiling was turned off for such regions. Profile information was also used by the speculation transformer to pack operations with narrow operands on a functional unit whenever the instruction-level parallelism exceeded the available functional unit resources. This dissertation introduces the concept of *heterogeneous packing*, which is a scheme to pack different types of operations on the same ALU [55]. This work differs from previous research that exploits operand width by packing similar types of operations in a SIMD-like fashion. The advantage of heterogeneous packing is that it exploits greater opportunities than by packing only similar types of operations together.

By exploiting the unused portions of the fetch unit, increases in fetch bandwidth and other hardware complexities were avoided. These unused fetch slots were used to fetch additional operations packed on a functional unit. The predictor component included a *steering logic* network to redirect operations packed together from the fetch buffer to their respective functional units (assigned by the speculation transformer). This steering logic extracted a subset of bits of each predicted operand value. The subset of bits was predicted by the compiler to be the maximum width of that operand. The operands of packed operations were verified using additional logic for detecting mispredictions whenever an operand's width spilled over the predicted width. In case of a misprediction, each mispredicted operation was re-executed independently on the ALU. From empirical analysis, bit-width prediction technique was found to be suitable for VLIW architectures constrained in the number of functional unit resources, such as some embedded processors.

1.3 Organization of this dissertation

The remainder of this dissertation is organized as follows. Chapter 2 presents background information on prediction-driven optimizations and profiling in the context of VLIW architectures. Prior research in value prediction in both superscalar and VLIW architectures is described here in detail. Previous work related to bit-width prediction is also described in this chapter. Chapter 3 gives an overview of the prediction framework. The components of the prediction framework are described in detail along with their interactions with each other during program execution. Chapters 4 and 5 present the framework implementation in context of value prediction along with empirical analysis on a VLIW machine model. Chapter 6 presents the framework applied to develop bit-width prediction with experimental evaluations. Finally Chapter 7 presents conclusions and future research directions.

Chapter 2
Background

Prediction-based optimizations for VLIW architectures is dependent on prior work in prediction methodologies, profiling techniques and compiler transformations. This chapter provides a background in these areas.

2.1 Prediction Methodologies

The idea of prediction has been a topic of research in processor and compiler design since branch prediction gained wide recognition in the early 1990s. Branch prediction has been successful in overcoming the latency of branches by prefetching instructions along the predicted path [51, 79]. Subsequently, several other prediction schemes have been proposed at the instruction-level for overcoming latency of memory instructions. One of these prediction schemes is address prediction [6] which predicts memory addresses of loads to overcome the effective address calculation latency. Load/store dependence prediction [15, 53] is used to predict aliases between load and store instructions. If a load instruction is predicted to be independent of prior stores, it is speculatively issued without waiting for prior stores to complete. Also, the prediction can estimate the store that a load is dependent upon. Memory renaming [75] is another form of memory prediction involving prediction of dependencies between loads and stores. The prediction involves forwarding of values from stores to the predicted dependent loads without accessing memory. Value prediction is a prediction technique that was initially developed to overcome the latency of load instructions [49]. In the context of superscalar processors, a comparison of the above prediction techniques that overcome the latency of memory instructions is given in [58].

Prediction techniques for VLIW architectures have been more restrictively applied due to the challenges discussed in the previous chapter. In this context, prediction is embedded in several phases of the compiler such as region formation [40, 50] and inlining [12], that use profile information to estimate the frequently executed blocks/paths. Memory prediction techniques have been implemented for VLIW compilers in previous work [32]. This previous work uses a hardware mechanism to store load/store dependencies and performs aggressive code scheduling in the presence of ambiguous memory dependencies. The hardware supports code reordering by detecting cases when the ambiguous references access the same location and subsequently invoke a recovery code sequence supplied by the compiler. The concept of recovery code has been used in VLIW compilers in other areas also, such as exception handling [3] and value prediction [28]. This dissertation explores the efficacy of static recovery code and also develops dynamic recovery methodologies that generate recovery code at run-time.

The prediction framework developed in this dissertation is applied towards optimizations that predict values and operand widths. The next few sections describe prior work related to these applied optimizations in detail.

2.2 Predicting Values

2.2.1 Methodologies

The concept of value prediction was introduced by Lipasti and Shen [49] and was originally targeted to reduce memory access time of load instructions. Computed values of load instructions were predicted in order to hide the memory latency. In their further studies [48], prediction was applied to all register defining instructions. Their studies show that programs exhibit a large degree of value locality. Using a simple history of the most recent value is able to capture 49% of the total run-time generated values. If the history is extended to four most recent values, this number rises to 61%. Their proposed value prediction unit used a table to store the most recently generated values for the instructions. These values were predicted to be the next computed value and the prediction scheme was called *last value prediction.* Their research was extended in work by Burtscher et al. [9]. This latter work implements several predictor configurations to predict the value of an instruction out of its last n computed values, where n ranged from two to eight. Confidence mechanisms were used to select the value from the set of previously computed values. The results from this work show that predicting from last four values outperforms the other predictor configurations scaled to the same size. Also the performance is comparable to other complex predictors of the same size. The authors suggest profiling to further improve the performance and reduce predictor size.

Gabbay and Mendelson [29] redefined the concept of value locality as *temporal value predictability*, occurring when an instruction computes a value as a function of the most recently generated value. They introduced another concept called *spatial value predictability*, where an instruction computes a value as a function of more than one previously generated values. This kind of predictability was measured on patterns of values that appeared as stride sequences and extrapolating the next value (*stride prediction*). Stride prediction was shown to outperform last value prediction, especially for ALU instructions in floating point benchmarks. A possible reason is that stride prediction was able to capture several loop induction variables which are usually ALU instructions. The difference between the two prediction mechanisms was not significant for integer loads and floating point instructions. Also, the improvement was noted to be higher in the computation phase than the initialization phase of floating point benchmarks. Classification of instructions was proposed using profiling to identify predictable instructions. Their observations suggest that different instructions in a program would have different potential for value predictability. And this potential seems to vary based on the prediction scheme being applied. In this dissertation, value prediction is implemented by statically identifying the predictability behavior of instructions and selecting the best prediction scheme(s) for each instruction.

Sazeides and Smith [64] introduced a new class of prediction which they define as *context-based prediction*. Last value and stride predictors were classified as being *computational predictors* performing some operation on previous generated values to predict the next value. Context based prediction stored finite value history patterns and predicted one of the values if the same pattern (context) repeated at run-time. The authors named their predictor a *finite context method* (FCM) predictor. The FCM predictor used the most recent history pattern to index into a prediction table and return the most likely successor to a given pattern. The authors observed that over 20% of the correct predictions made by the FCM predictor could not be captured by last value or stride prediction. Also, only 20% of the static instructions accounted for almost all of the improvement of FCM over last value and stride prediction. Similar to other previous work, the authors noticed certain instruction types to be more predictable than others. Although this study used unbounded history table sizes, the authors justified implementing context-based prediction from their observation that the majority of the instructions (over 90%) generated few distinct values (less than 4096). Also, since only a small fraction of static instructions contributed to improvements over previous predictors, these instructions could be captured by profiling and used for prediction in the context of VLIW processors. Using the profile information, a predictor can be designed that performs FCM prediction only on this fraction of instructions, while performing last value or stride prediction on other fractions of instructions.

Several hybrid predictors have also been developed to accumulate the benefits of different prediction schemes into one hardware module. Wang and Franklin [76] investigated a two-level scheme to perform value prediction. The first level stored patterns of values generated by the instructions along with the four most unique values, and was accessed by an instruction's address. The selected pattern was used to access another table that stored the history of previous outcomes of different patterns. A selection criterion operating on the history chose one of the four values as being the next prediction. This scheme was based on the authors' observation that a substantial percentage of the dynamic instructions have four or fewer unique values. The authors further proposed a hybrid scheme involving the two-level and the stride predictors. Rychlik *et al.* [59] designed hybrid predictors composed of stride and FCM predictor components. Run-time performance was used to determine which component predictor to access and update during a prediction. These predictors produced high prediction rates. However this improvement was at the expense of large prediction tables. The prediction mechanism used assumed the table updates were immediate and ideally correct. Realistic updates reduced the prediction rates significantly (24% for the smaller-sized predictor configuration). Another hybrid predictor design that uses last value, stride and two-level prediction schemes appears in [46].

2.2.2 Reducing Hardware Cost for Value Prediction

The above hybrid schemes improve performance over individual prediction schemes by applying each of the individual schemes on an instruction. However these schemes are expensive in terms of hardware cost since each prediction scheme needs to be completely implemented. Most of the above schemes use a separate table, a classification table, to analyze the most recent values for each instruction and decide whether to predict the instruction or not. In case of hybrid predictors, the classification table also decides which prediction scheme to use whenever an instruction is selected for prediction. This run-time classification requires extra logic in addition to the value predictor hardware, and each predictable instruction involves a learning period before it can be selected for prediction. Spurious mispredictions can evict an instruction from the table of predicted values and this instruction needs to go through the learning period again before it can be predicted. Another design overhead in hybrid value predictors is the cost of accommodating different prediction schemes. To achieve high prediction accuracies, hybrid predictors need to be designed with the cost of storing several different types of prediction histories for each predicted instruction. Even with high prediction accuracies, the usefulness factor reduces the performance benefit of these predictors. Previous work suggests a usefulness tracking mechanism to update a value predictor only if a prediction is read by an instruction [59]. This mechanism results in slight improvements in speedups but reduces overall prediction accuracies. The observation that even a reduction in prediction accuracies can still improve performance, indicates the significance of the usefulness factor in prediction.

Some researchers have attempted to optimize the predictor cost by reducing the stored prediction history. Morancho *et al.* [52] propose a technique for load address prediction that avoids placing unpredictable loads in the predictor table. The technique stores some bits of data addresses generated by each load in the classification table. Consecutive data addresses of each load are compared using the stored bits and if they are the same, the load's entry is placed in the predictor table. Only a few bits of each data address need to be stored. Though this scheme is applicable for last address prediction, last value prediction would require a larger number of bits for comparison, since the range of possible data values is much larger than the range of generated addresses. Also the comparison would be more complex for prediction schemes other than last value. Rychilik *et al.* [60] avoid unpredictable instructions by evicting low confidence instructions from their hybrid predictor table. An evicted instruction is either never predicted again or classified for prediction using another scheme, depending on which scheme was currently used for prediction. In the former case, the temporal confidence measure may be too harsh in deeming an instruction to be unpredictable. In the latter case, the re-classification may yield the same prediction scheme as before. Sato [62] attempts to reduce the cost of predictor tables by exploiting narrow width values. The predictor table is partitioned into a wide partition and a narrow partition. This dissertation reduces the cost of the predictor table using profiling and usefulness analysis to eliminate the predictions with low potential for improving performance. This potential is measured from the accuracy and estimated usefulness of the predictions.

Tullsen and Seng [73] avoided the use of buffers to store potential values for prediction. They assumed the result of an instruction may already be available in the register file. The study was based on the observation that 75% of the time a value is loaded from memory, it is already in the register file, or was recently available. In order to detect such reuse of register values, compiler transformations as well as hardware techniques were proposed in this work. In order to exploit correlated values across different registers, the compiler would assign the same register to multiple values. In case reuse occurred across the same register, hardware was used to exploit it. Profiling was done to mark the instructions as candidates of either same-register value reuse or reuse across different registers. Also both static and dynamic prediction techniques were investigated. The performance of the static prediction techniques displayed a severe penalty of mispredictions on the results. The proposed techniques were similar in performance to the simple predictors such as last value, with the advantage of no hardware cost of storage of values. Their work indicates the potential of the compiler in assisting the value prediction process and reducing the associated hardware costs.

Calder *et al.* [11] examined techniques to use value prediction selectively in the presence of hardware constraints such as prediction table capacities and misprediction penalties. They evaluated different strategies for initializing and updating confidences of instruction values, as well as replacement strategies during capacity misses. Also the authors attempted to estimate the critical path at run-time by keeping track of the longest dependency chain. Instructions on such a path are selected for prediction as well as for value speculation. In related work [74], the same authors observed higher speedups when instructions on such paths are speculated, even when these instructions have lower confidences than the off-path ones. For the research in this dissertation, instructions for prediction are selected along the paths that are significant to the program's performance such as critical and frequently executed paths.

Muth *et al.* [54] use a different approach to avoid value predictors by using code specialization. In their work, value profiles are used to identify program points where specialization may be profitable. The value and expression profiles are obtained for those program points and used to carry out specialization. The benefit of predicting an operation is estimated as the number of cycles it takes to execute that operation and does not involve any sophisticated usefulness analysis. Although the performance improves, it is affected by increase in cache miss rate and mispredictions due to static code growth.

2.2.3 VLIW Processors and Value Prediction

In context of VLIW processors, Fu *et al.* [28] proposed a scheme to perform value prediction for VLIW processors. Instructions were predicted statically and used to speculate instructions during instruction scheduling. The instruction set was extended to use explicit instructions for predicting values and verifying the predictions. The value prediction hardware was updated through software during execution of the new instructions. Mispredictions were handled by statically scheduling code that would execute for recovery. For each prediction, an additional branch to the recovery code was added and instructions between the prediction and the branch were made candidates for speculation based on the predicted value. The experimental framework considered all load instructions for prediction. Although the compiler controlled the degree of prediction and speculation, the prediction techniques themselves remained dynamic. The predictors incorporated in their study included the two-level and hybrid predictor of Wang *et al.* [76] but aliasing effects were not considered in the predictors. A software-only version of their work [27] avoids any hardware extensions by using available registers to store predicted values. Also, the Instruction-set Architecture (ISA) extensions were avoided using move and add instructions for last value and stride prediction. However, the performance of the software-only version does not measure up to the prediction model that uses hardware and ISA extensions. In their subsequent work, mispredictions are handled using interrupt handling routines and register renaming techniques [26]. The authors also analyze different heuristics for selecting the operations to predict. One of the drawbacks of software-controlled value prediction is the overhead of the branches. Larson *et. al* [43] propose to reduce this overhead by using a branch predictor to gauge the confidence of a prediction.

The research in value prediction developed in this dissertation is most closely related to the previous work by Fu *et al.* [28]. Besides using the profiler for filtering instructions for prediction, this dissertation also uses it for designing efficient predictor tables. Dynamic generation of recovery code is also designed to avoid the problems associated with static recovery, such as code growth and additional branches.

2.3 Predicting Operand Widths

As processors have shifted to wider word widths and with multimedia processing become more prevalent, several processors have added support for handling small operand widths. These processors have enhanced their instruction set with operations for sub-word parallelism (SIMD instructions). Examples of these instructions sets include Intel MMX and SSE, AMD 3DNow!, Motorola Altivec, and HP MAX-2. Although these enhanced instruction sets encode sub-word parallelism in the form of SIMD instructions, most of the burden of detecting the parallelism is left to the programmer. Vectorization techniques have been proposed in the context of vector processors to exploit loop-level parallelism within scientific code [20, 45]. Several benchmarks are either difficult to vectorize or are partially vectorizable. In the case of applications running on fine-grained architectures, there has been recent work in hardware and compiler support for synthesizing SIMD-like instructions without user intervention. Brooks *et al.* [8] proposed an architecture that dynamically packs narrow integer operations on an FU, similar to a parallel sub-word operation. This previous work uses dynamic packing to reduce power consumption as well as to improve performance. Power is optimized by using aggressive clock gating to turn off portions of integer arithmetic units that are unnecessary for narrow bit-width operations. Packing is applied to improve performance by merging narrow integer operations to share a single functional unit. The optimization for power consumption reduces the integer unit's power by over 50% and the performance oriented optimization improves speedups in the range 4-10%.

Compiler support to synthesize SIMD instructions has been proposed by Larsen *et al.* [42]. This previous work introduces a concept known as Superword Level Parallelism (SLP) that detects operations of similar types in a basic block and executes them in parallel by packing them together into a SIMD instruction. Packing of adjacent memory references was found to be responsible for most of the performance benefits of SLP since it eliminates individual load/stores and address calculations. The overhead of packing memory references is the packing of data for executing the SIMD instruction and any unpacking that may be required after execution. In a separate study, bit-width analysis [72] has been proposed to reduce the silicon area of a reconfigurable processor. The analysis examines each static instruction in a program to determine the narrowest return type that retains program correctness. For each variable, a data-range is associated which is propagated over a program's control flow graph. Information extracted from bit-width analysis is used to reduce the number of bits necessary to represent each program operand. Initial experiments anticipate reduction in logic area by 15-86% and power reduction by 46-73%.

These previous studies are oriented toward superscalar processors. This dissertation exploit smaller operand widths for VLIW processors using profile information.

2.4 Profiling

Fine-grain profiles of basic blocks and control-flow edges have become the basis to guide optimizations in compilers. One use of profiling is to identify the frequently executed paths in a program. Path profiling algorithms have been developed to determine how many times each acyclic path in a procedure executes. Ball and Larus [5] developed a profiling algorithm using a spanning tree to determine a minimal low-cost set of edges to instrument. Young and Smith [80] used program tracing to record paths for performing branch correlation.

Previous work by Sato [61] has proposed the use of profile information to statically choose between memory renaming and address prediction for an instruction. In case profiling indicates a load value to be more predictable than predicting its data address, the value is predicted by forwarding results from aliased store instructions. Otherwise a stride predictor is used to predict the generated data address. In this previous work, each instruction is statically assigned the more suitable prediction scheme out of memory renaming and address prediction, and cannot change this assignment at run-time. Usefulness of a prediction is not considered and prediction history for all instructions is stored. Also memory renaming is used for predicting load values. However studies have observed that renaming is not as effective as value prediction in terms of speedups [58].

Gabbay *et al.* [30] were the first to propose profiling of data values for value prediction by the compiler. Value profiling was performed to insert opcode hints within the instructions, classifying whether an instruction is predictable, and if so, whether it is predictable using last value or stride. Profiling was also used to eliminate the stride for instructions that can be accurately predicted using last value. The potential of profiling was compared to the dynamic scheme of classification using saturating counters. The authors observed that in most benchmarks, the profiling-based classification better eliminated mispredictions. However the number of correct predictions decreased for most of the benchmarks as compared to saturating counters. The explanation provided was that most of the benchmarks employed small working sets of values and could not exploit the benefits of static classification. This dissertation incorporates more complicated techniques for value prediction such as FCM and path-based prediction in the profiler. Having several prediction schemes alleviates the problem of less prediction due to static classification, by capturing more instances of prediction. Also, in an attempt to capture smaller working sets of values, this research performs value prediction selectively, such as predicting only along specific paths of execution.

Calder *et al.* [10] proposed techniques to perform value profiling in superscalar processors. Profiling of load instructions was performed and the most frequent N values for each instruction were recorded. Among their observations, the authors find a table size between 4 and 6 to be sufficient to accurately capture the invariant information of an instruction. The authors also proposed a technique to profile with less overhead, known as *Convergent value profiling*. In this technique, profiling of instructions was relaxed by instrumenting less frequently in case the invariance for an instruction was observed quickly. Another technique that reduces profiling cost has been suggested in [78]. In this latter work, the profiler uses a cost-benefit analysis to selectively instrument the program points for profiling. The analysis involves simple probability tests at link-time estimating the cycle improvements due to prediction and does not involve dataflow information. In this dissertation, usefulness of predictions is used to reduce the profiling cost. The usefulness analysis developed for optimizations is found to be quite accurate since it is statically performed and involves program dataflow information.

2.5 Predictability Characteristics

There has been some work that attempts to understand the predictability of instructions. Sazeides *et al.* [65] presented a model to analyze value and control predictability. Predictability was studied similar to dataflow analysis in terms of generation, propagation and termination. The authors observed that most of the predictability could be traced back to program control structure and immediate values. Program input data was found to be a relatively unimportant source of predictability. Predictability was usually terminated by unpredictable memory data or when a correctly predicted value combined with incorrectly predicted value in an instruction. One of their observations was that a majority of mispredicted branch instructions (more than 50%) have predictable input values.

Rychlik *et al.* [59] observed that a large chunk of value predictions were never used by any dependent instructions. On average, 31% of SPECint95 and 62% of SPECfp95 integer predicted values were never used. The authors reduced such useless predictions by updating the prediction tables with the correct value only if any dependent instruction used the predicted value. The percentage of useless predictions were reduced significantly because of this monitoring. It was observed that predictions for instructions closely followed by their consumers benefited most from the monitoring. This dissertation uses the concept of usefulness of predictions as part of the conceptual prediction framework.

Predictability of instructions has been quite successful in improving performance because there is a lot of redundancy in programs. The redundancy is because of the repetitive behavior of program instructions. A study that analyzes instruction repetition observes that most of the repetition is due to program internal values and global initialized data [70]. Significant repetition is also seen due to function prologue and epilogue. Besides predicting values, another way to exploit this repetitive behavior is to reduce the number of executing instructions by providing microarchitectural support for buffering previously computed values. Future dynamic instances of the same instruction can use the buffered value if the input operands in both cases match. This technique is known as *Instruction Reuse* [69] and does not involve prediction. Instead, the execution of an instruction is bypassed whenever the result is available in the hardware buffer thereby reducing resource utilization and overcoming execution latency. A previous study [71] has compared value prediction and instruction reuse and observes that performances are comparable. Although instruction reuse captures less amount of redundancy, the early validation of results has a higher payback than from prediction. Exploiting redundancy through reuse has also been explored at more coarse granularities such as the basic-block level [38] as well as trace level [33]. Instruction reuse has also been investigated using the compiler for identifying code whose computation are reusable during execution [17]. This previous work attempts to eliminate dynamic computation by creating reusable code segments, The work is extended to select the regions dynamically by run-time monitoring [18].

2.6 Scheduling and Code Transformations

Several techniques to perform scheduling using code motion to increase instruction level parallelism have been developed. These techniques include Trace Scheduling [24], Percolation Scheduling [2] and Region Scheduling [36]. Each technique considers different approaches to identify candidates for code motion. Trace Scheduling creates large blocks, called traces, consisting of several basic blocks along a program path that is estimated to be frequently executed. Instructions appearing within a trace are reordered to generate a good instruction schedule for the trace. This schedule may be generated at the expense of overhead in the compensation code inserted in the off-trace blocks that are not very frequently executed. Percolation scheduling performs code motion using a set of primitive program transformations. These transformations are applied in a hierarchical manner to exploit both fine-grained and coarse-grained parallelism by globally rearranging code to exploit parallelism. Region Scheduling operates on a Program Dependence Graph (PDG) and divides a program into regions containing statements having the same control conditions. The region scheduler performs transformations on the PDG to redistribute excess parallelism through code motion from one region to another region with insufficient parallelism.

Code transformations are widely used to increase the instruction-level parallelism for VLIW processors. Some transformations are performed by predicting certain program attributes such as frequently executed paths. Gupta [35] performed global code motion with the help of path profiles. Some researchers have addressed transformations to enhance other program optimizations. Bodik *et al.* [7] proposed slicing transformations to perform partial dead code elimination. This dissertation uses the prediction framework to develop optimizations that are applicable either as scheduling techniques or as separate code transformations. In particular, value prediction is developed as a separate transformation phase that allows the scheduler to aggressively speculate instructions. On the other hand, bit-width prediction is developed as a scheduling technique that performs operand width-sensitive scheduling.

Chapter 3
Prediction Framework

The framework components for prediction-based optimizations and how they interact with each other during program execution are described in this chapter. The framework is designed to be general enough to enable the application of various prediction-based optimizations on a VLIW architecture. The functionality of each framework component is described here and subsequent chapters incorporate this functionality to develop specific optimizations.

3.1 Overview of Framework

The framework design provides a model that allows a methodology for predicting instruction attributes to be translated into a code optimization. Although the functionality of the components is presented here at a conceptual-level in the framework, the implementation of the specific optimization being applied can involve additional issues. A high-level view of the framework structure is shown in Figure 3.1. This figure shows the framework components (displayed in shaded) interacting with program execution. The prediction schemes serve as input to the framework and are used to help design parts of the framework. Based on the optimization being applied, different schemes can be designed. For example, memory addresses are predictable using stride prediction due to a large percentage of memory data structures traversed iteratively in programs. On the other hand, stride prediction does not work well for predicting values of memory instructions [29] and more sophisticated prediction schemes need to be devised. The prediction schemes are developed initially and their design affects the implementation of framework components.

Within the framework, the *usefulness analyzer* incorporates static analysis to identify program regions where prediction would affect the instruction schedule to improve overall performance. This framework component uses the program dependency information extracted by the compiler front-end to estimate the benefits of the prediction on program

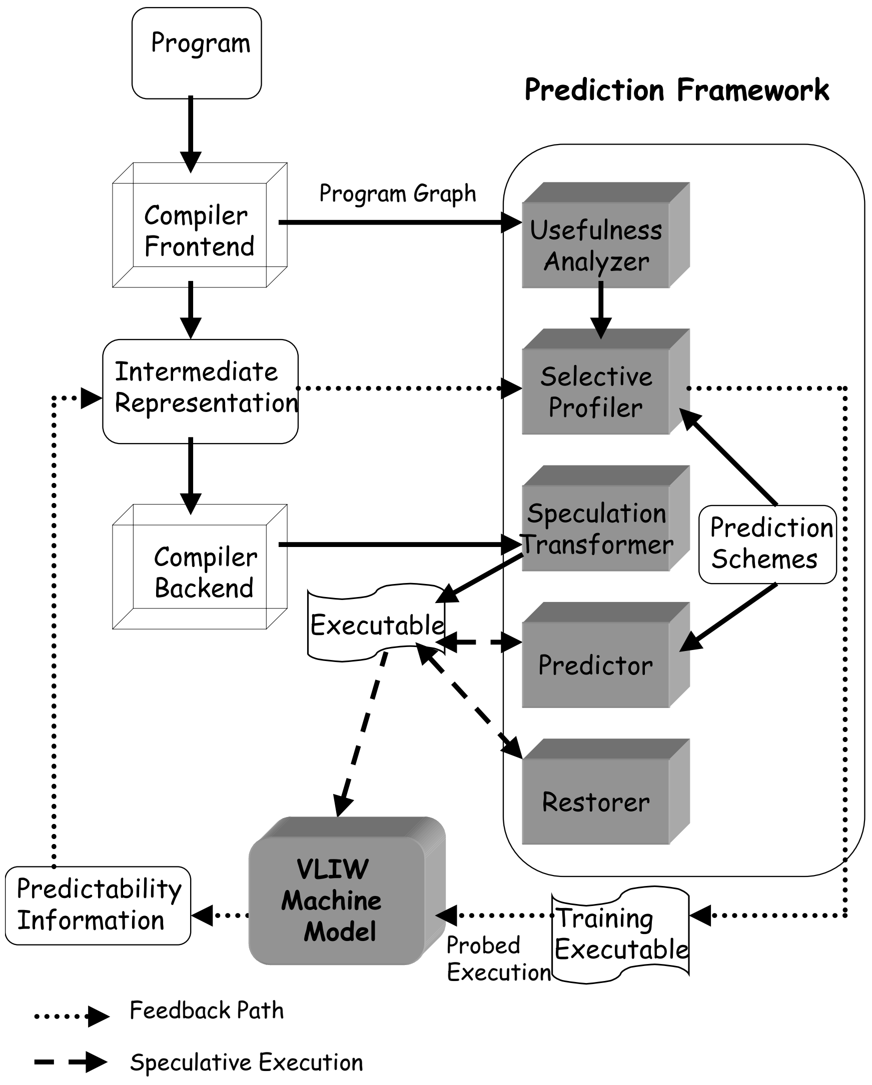


Figure 3.1: High-level view of the Prediction Framework

regions. This analysis can be applied at the instruction-level or at a region-level such as basic or extended blocks, depending on the optimization and the cost of analysis. The *selective profiler* extracts run-time information to gauge predictability of the program regions deemed useful by the usefulness analyzer. The profiler uses the prediction schemes comprising of techniques for making predictions on the instructions. The measurement of predictability is performed by instrumenting the program and generating a probed executable (labeled as training executable in Figure 3.1). The instrumented code analyzes predictability of instructions using the designed prediction schemes. This executable is run with training inputs, and profile information is generated during this initial phase of execution. The generated information annotates the intermediate representation of the program. The *speculation transformer* uses the annotated representation to statically select the instructions predictable from the profile data and reorder code at compile-time allowing speculative code motion. The speculation transformer works in conjunction with the compiler back-end using machine-specific information to generate code incorporating speculation. At run-time, as the program executes on the VLIW hardware, the *predictor* enables predictions to be made dynamically. The *restorer* handles run-time mispredictions by re-executing the instructions that were executed incorrectly due to a (mis)prediction.

Among the framework components, the usefulness analyzer, selective profiler and speculation transformer are active at compile-time and the predictor and restorer interact with program execution dynamically. The prediction schemes are devised initially before implementing the rest of the components of the framework for an optimization, as they are used in implementing some components. In the case that the history of previous computations is necessary to make future predictions, this history needs to be captured at run-time in the predictor. In this case, the design of the predictor is based on the underlying prediction schemes. Also, these prediction schemes are used by the selective profiler to test program predictability. These relationships of the schemes with the framework components are shown in the Figure 3.1.

One of the features of the framework design is that the framework components interact with each other. The framework components span the domains of compilers, profiling and hardware. Information in one domain can be useful to **reduce the cost** associated with a component designed in another domain. This kind of synergism also works well to **increase the benefit** of the framework components beyond the limitations of the domain in which they are applied. As an example, profile-guided transformations typically utilize global characteristics of run-time information and are unable to react to phased behavior of programs. As mentioned in the introduction, phased behavior is the tendency of a fragment of code to exhibit a sequence of temporal behaviors, some of which are easier to predict than others. In context of the prediction framework, some phases of an instruction's execution are easier to predict than others. The framework can address phase changes by designing a predictor that adjusts the prediction scheme dynamically to adapt to the phase changes. This hybrid design increases the benefit of the speculation transformer, allowing more predictions to be used for aggressive speculation. However, the cost of designing such a hybrid predictor can be a significant factor. This cost can be reduced by performing feedback analysis of run-time predictability behavior of benchmark programs and using this analysis to optimize the predictor design. The idea is to use the profiler to identify the most suitable prediction schemes that should be combined into hybrid schemes. Feedback analysis would identify the most frequent phase changes and tune the hybrid prediction schemes towards these phases and their corresponding prediction schemes.

The next section describes the prediction schemes as well as the framework components in detail.

3.2 Framework Description

3.2.1 Input Prediction Schemes

As mentioned in the background chapter, several different prediction schemes have been developed to predict various attributes of program instructions. Recently, there has been some work that evaluates and compares different prediction techniques [58]. This study finds that it is possible to accurately predict which memory operations would access the same location, with almost 99% of memory locations being predicted accurately. On the other hand, less than 50% of the load instructions have their addresses correctly predicted. This observation suggests that certain instruction attributes can be predicted highly accurately using existing schemes while others require the design of more efficient prediction schemes for higher accuracies. As mentioned before, the prediction accuracy of a characteristic for an optimization is especially a significant issue in VLIW architectures due to the additional challenges of the in-order execution model preventing the execution of subsequent instructions upon a misprediction. Hence a prediction-based optimization should be applied only if speculation is correct the majority of the times it is performed. In the framework developed in this thesis, one of the first steps in implementing a prediction based optimization is to have effective prediction schemes steering the optimization. This capability is achieved by either designing novel prediction schemes or using existing schemes that are able to predict the attribute used for speculation with reasonable accuracy. These prediction schemes serve as a pre-requisite for developing the rest of the components of the framework. By designing the prediction schemes initially, the framework components are tuned according to these schemes. For example, if a stride prediction scheme is found to be useful for an optimization, the predictor should be implemented to predict stride values for making predictions. Also, the profiler should be developed to analyze the accuracy of stride prediction. As mentioned before, the efficacy of prediction schemes can vary differently for different optimizations. For an optimization, the most effective schemes are identified and considered for implementing the framework.

3.2.2 Usefulness Analyzer

There has been substantial research in building accurate predictors that allow speculation of instructions using the predictions. Lately, some researchers have observed that predicting all candidate instructions may not always improve performance [59, 11]. The reason is that several of the predictions are not used before the actual computation (being predicted) is evaluated. Hence, if computed values were predicted aggressively (similar to the example prediction of Figure 1.1), a majority of those predictions would not result in any of dependencies being speculated. Although this previous study refers to a superscalar processor, it is applicable to the VLIW architecture as well. In the latter case, even though a prediction generates speculation, performance improvement may be inhibited due to conservative static scheduling. The VLIW compiler explicitly schedules those instructions that cannot be speculated, to avoid using the predicted value. For example, stores cannot be speculatively executed since they may alter the machine state incorrectly. In Figure 3.2, assuming a single cycle for verification, predicting the value of r_1 would not improve schedule length, even in the case of correct prediction because the store instruction is non-speculative and needs to be scheduled after verifying the prediction. Similarly, anti-dependencies that may alter a predicted value either need to be scheduled to avoid the speculation or need to be renamed. Hence only *useful predictions* that can potentially improve the schedule need to be considered for prediction.

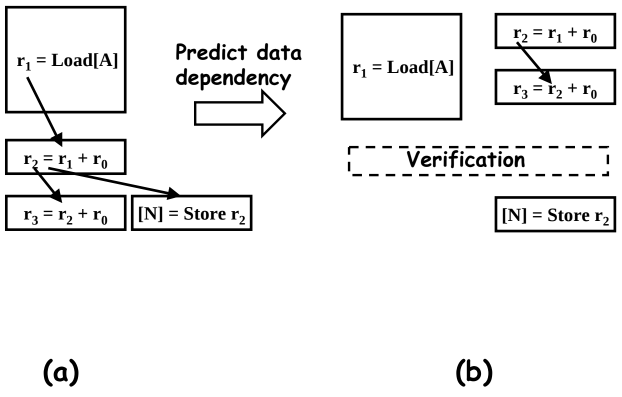


Figure 3.2: Example of Useless Prediction

The predictions that do not improve the instruction schedule can also be harmful for program performance. These predictions can degrade performance even when accurately predicted due to the additional verification required after predicting. Also the overall prediction accuracy reduces in such cases when run-time history is used for making predictions. Run-time history is typically stored in the predictor and accessed by an instruction using indexing schemes, such as the program counter. By storing the history of predictions that do not improve the schedule, the history of the useful predictions can get corrupted due to aliasing effects in the predictor. This effect is more pronounced in predictors of smaller sizes. Hence it is imperative to consider usefulness of predictions in the VLIW domain.

In the prediction framework, useful regions are identified by estimating the benefit of applying prediction using a two-step approach. Initially the useful basic blocks are selected by identifying the frequently executed blocks using control-flow profile information. Instructions within useful blocks are analyzed for usefulness using static analysis. An instruction whose prediction results in improving the instruction schedule is marked as being useful. Since the underlying machine uses a static scheduling model, the estimation of usefulness can be designed to be quite accurate using dataflow analysis. While developing the usefulness analyzer, the cascaded effect of predictions on schedule needs to be considered. For example, in Figure 4, the effect of predicting the value of r_1 will not be useful unless the value of r_2 is also predicted.

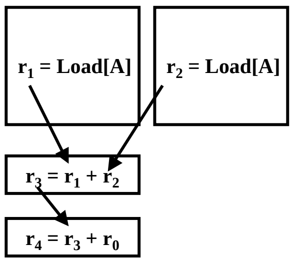


Figure 3.3: Example of Speculation by predicting multiple loads

Hence, the usefulness analyzer estimates the benefit of predicting an instruction on the performance, and accurate analysis is necessary for correctly estimating this benefit.

3.2.3 Selective Profiler

Profiling has been traditionally used to gather run-time statistics in static scheduling compilers. There are three ways of collecting profile information from a program: *sampling*, *instrumentation* and *simulation*. The sampling-based method is based on techniques that generate interrupts at regular intervals of program execution. The profiling routine then logs the execution state of the program at the time the interrupt occurs. Instrumentation inserts additional code to call the profiling routines upon execution of every region being profiled (a region can be an instruction, a basic block or a procedure depending on the profiling granularity). The third profiling technique, simulation, collects execution statistics by running the program on a simulator that monitors the program performance. Amongst the three techniques, instrumentation and simulation technique are complete because both cover all the regions that are being profiled. On the other hand, sampling is faster since it does not involve additional instructions to be executed along with the program. However sample-based profiles are incomplete since it is possible for instructions to be executed without being sampled. The framework designed in this dissertation covers prediction at the instruction-level. A sampling-based profile system cannot provide accurate instruction-level information because it is incomplete. Instead, simulation and instrumentation were used for profiling in the framework. Chapter 5 elaborates on these techniques as profiling tools in the context of a compiler infrastructure used for experimentation. Also a description is provided for the conditions under which each of the two profiling tools is chosen.

The cost of profiling for providing feedback to the compiler can become a significant overhead by profiling at the instruction-level. In the context of prediction, the cost of profiling is even more of a concern since different schemes may be necessary to test the predictability of an instruction. In order to select an instruction for prediction, the profiler may need to apply multiple prediction schemes to check whether the instruction is accurately predictable. Different instructions may be predictable using different prediction schemes. Hence, the profile overhead can become substantial if multiple prediction schemes are applied to each instruction.

The cost of profiling is mitigated in the framework by using the information extracted from the usefulness analyzer to discriminate instructions for profiling. The selective profiler marks instructions identified as being useful to estimate their predictability. Based on the type of prediction being implemented, different attributes can be profiled, such as computed values, memory addresses, bit-widths, etc. Also, multiple prediction techniques can be tested as long as these techniques can be applied at run-time by the predictor framework component. The predictability is measured by applying different prediction schemes (developed beforehand) on the data being profiled, and computing their prediction accuracies. The program intermediate code is annotated with this prediction information.

3.2.4 Speculation Transformer

The speculation transformer modifies the program by statically speculating instructions using code-restructuring transformations. The profile information is used to select instructions that meet the predictability threshold. Speculation may require designing additional machine instructions to indicate the instructions being predicted and/or speculated. These semantics are necessary to specify to the predictor at run-time which instructions use predicted values. Also, based on these additional semantics, the restorer would select which instructions to re-execute upon mispredictions.

This framework component can be provided as a separate optimization phase in the compiler to expose additional parallelism to subsequent optimization phases. Another option is to integrate the speculation transformer within the instruction scheduler. An example of the former case is the example of Figure 1.1 where value prediction removes some data dependencies exposing greater ILP to the scheduler. Figure 1.2 showed an example of the latter case. In this example, the instructions with narrow operands were marked as 16-bit operations by the profiler. This information was used by the scheduler to pack such operations on the same ALU. The speculation transformer was implicitly incorporated within the scheduler by adding bit-sensitive information to the scheduling algorithm. In either case, the speculation transformer utilized the profile information to speculate instructions on the execution path of the predictions.

3.2.5 Predictor

Although the framework exploits predictions by compile-time scheduling, the predictions are actually made at run-time. The prediction framework component provides the support to uncover the predictions at run-time. Incorporating predictions modifies the execution pipeline due to the additional functionality of predicting and verifying values. The predictor component includes hardware logic to perform these functions. The actual implementation of the predictor depends on the attributes being predicted. As mentioned before, histories of predictions are widely used in prediction-based optimizations to help make future predictions. For most prediction-based optimizations, hybrid prediction schemes are more accurate than an individual prediction scheme. For example, while predicting effective addresses of loads, a stride predictor can predict addresses that are a fixed stride apart, such as addresses of consecutive nodes of a fixed data structure (e.g. array). On the other hand, a context-based predictor bases its prediction on the last several values seen and can recognize repeated values without a fixed stride, such as addresses of elements of a linked list. A hybrid address predictor combines the capability of both these schemes and predicts more accurately because of its flexibility to dynamically choose the best scheme [6]. In the cases when prediction history is used, the predictor design may also include additional buffer storage of the prediction history. Depending on the amount of history that needs to be stored, it may be possible to use existing registers for this storage [73]. The design of the input prediction schemes controls the decision of how much additional buffer is needed for making accurate predictions. In the case when an additional buffer is necessary, the predictor design also involves addressing issues such as hardware complexity and aliasing.

The predictor design can be optimized by applying feedback information about the behavior of programs in terms of predictability for the underlying optimization being developed. This information is useful to reduce unnecessary complexities within the predictor. For example, if a simple prediction scheme works for several of the prediction cases, the predictor need not apply complex hybrid predictors to all predicted instructions. The compiler can identify the instructions that suffice with simple prediction schemes as opposed to those requiring hybrid prediction. Also, feedback analysis can be used to tune the hybrid scheme by selecting the individual schemes most suitable for the hybrid design. This feedback analysis can be performed after developing the input prediction schemes and observing their performance by profiling benchmark programs. This design is a one-time process during the construction of the framework. Conventionally, run-time history is stored in the prediction buffer and is accessed by instructions using the program counter. Since predictions are made statically in the framework developed in this dissertation, the compiler can be used to assign the predictions to the buffer to reduce aliasing effects. The predicted instructions can be assigned buffer entries to reduce the overlapping of instructions that are live at the same time.

3.2.6 Restorer

The restorer manages mispredictions at run-time and restores the program execution to a correct state. After detecting a misprediction, the instructions that were mispredicted need to be re-executed with the verified value of the prediction. This recovery costs additional run-time overhead but is necessary for correct program execution. Traditional approaches in compiler-controlled speculation have performed recovery during incorrect speculation by use of static recovery code [3, 28, 32]. The compiler is responsible for generating additional code corresponding to each prediction, called recovery blocks. The instructions inserted in a recovery block consist of all the instructions speculated from a prediction in the original (home) block of the compiled code. The home block is modified with an additional check that verifies the prediction and if the check fails, control flows to the recovery block. Speculative instructions belonging to the home block are non-speculative in the recovery block. The reason is that during execution of recovery code, program control has already ensured that these instructions should execute (with the verified value of the prediction).

One of the overheads of static restorer design is the increase in static code size, and this overhead becomes substantial with aggressive speculation. For example, the average code growth for safe control speculation in presence of exceptions by incorporating recovery blocks is 19% on average [3]. Static code growth can also affect instruction cache performance, increasing cache conflict misses with the normally executing code. This effect would be especially noticeable when prediction accuracies are not high, as in the case of instructions with phase changes. Another overhead of the static restorer is that the recovery code stalls the execution of the original code. Due to the lack of any parallel execution between these two code forms, the original VLIW code would wait for any recovery to complete whenever a misprediction occurs.

It is also possible to design the restorer in the prediction framework to involve an interplay between compiler and hardware. This design adds hardware support that generates recovery code at run-time. The advantage of the dynamic restorer is that static code growth is prevented. Also recovery can be performed in parallel with original code without stalling the VLIW engine upon a misprediction. This concept is similar to the DIVA checker architecture [4], that verifies the correctness of a superscalar core processor's computation. The verification in DIVA is performed by augmenting the commit phase of the processor's pipeline with a functional checker unit that checks the correctness of the core processor computations. The design of the restorer is based on similar principles of run-time verification of speculation in the context of prediction. This design may require additional functional units or use existing units to interleave recovery code with original executing code. In either case, additional architectural capability would be required to synchronize the execution of the dynamic restorer with the VLIW execution engine. This synchronization also needs compiler support to insert semantics for correct register communication between recovery code and the original code.

The above components were used to develop value and bit-width prediction. Value prediction is discussed in Chapters 4 and 5. Chapter 6 discusses bit-width prediction.

Chapter 4
Value Prediction Schemes

Various methods for value prediction have been proposed to overcome the limits imposed by data dependencies within programs. Previous value prediction schemes have focused on a **local context** by predicting values using the values generated by the **same** instruction. This dissertation develops techniques that predict values of an instruction based on a **global context** where the behavior of **other** instructions is used in prediction. The global context is captured by using the path along which an instruction is executed to predict its value. The techniques developed augment conventional last value and stride predictors with global context information. These techniques along with previous techniques were used as input prediction schemes to the prediction framework to estimate the efficacy of applying value prediction.

Previous studies performed to observe the patterns of values computed by an instruction have found that instructions demonstrate locality of data values [49]. The sequences that occur in certain patterns are easier to predict than others. Typically, the patterns of values include:

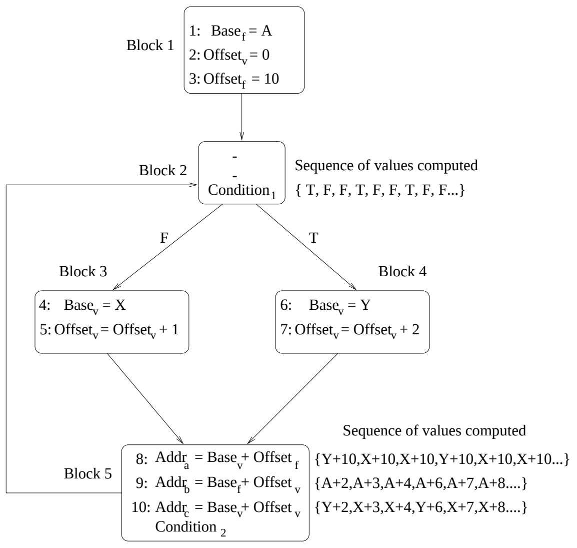
1. **(1) Constant values** - these are the easiest to predict by always predicting the same constant value. Constant values have been shown to occur often in benchmark programs [49].

2. **(2) Values differing by a stride** - these usually occur in case of loop induction variables or variables that are dependent on induction variables.

3. **(3) Other sequences** - these include sequence of values that differ by non-constant strides. This category is the hardest to predict.

The techniques that perform prediction based on a local context of an instruction include last value prediction in which the value predicted is the one that was computed in the last execution of that instruction. Stride prediction is another local context method that estimates the stride between consecutive values based on the previous execution instances of the instruction. Last value prediction can be viewed as a special case of stride prediction when the stride value is zero. Finite context method (FCM) prediction [64] and other two-level prediction [76] schemes are also examples of local context prediction based on previously observed patterns of values. These techniques attempt to detect sequences that are composition of stride and non-stride sequences.

This dissertation develops techniques that associate a more global context with an instruction to help predict its value. This global context association is analogous to branch prediction performed by correlating with other branches [79] and has not been considered for value prediction before. The global context considered is the path information of execution. Different values for an instruction are predicted along different paths within a program. Branch history is used to separate the different paths. In the work by Lipasti[47], branch history was used to either select instructions for prediction or predict input operands for estimating data dependencies. However, branch history was not involved in the actual prediction of computed values.



Block 1
1: Base$_f$ = A
2: Offset$_v$ = 0
3: Offset$_f$ = 10
Block 2
-
-
Condition$_1$
Sequence of values computed
{ T, F, F, T, F, F, T, F, F...}
F
T
Block 3
4: Base$_v$ = X
5: Offset$_v$ = Offset$_v$ + 1
Block 4
6: Base$_v$ = Y
7: Offset$_v$ = Offset$_v$ + 2
Sequence of values computed
Block 5
8: Addr$_a$ = Base$_v$ + Offset$_f$
9: Addr$_b$ = Base$_f$ + Offset$_v$
10: Addr$_c$ = Base$_v$ + Offset$_v$
Condition$_2$
{Y+10,X+10,X+10,Y+10,X+10,X+10...}
{A+2,A+3,A+4,A+6,A+7,A+8....}
{Y+2,X+3,X+4,Y+6,X+7,X+8....}

Figure 4.1: Example of Path-sensitive Prediction

Consider the code sequence shown in Figure 4.1. Instructions 8, 9 and 10 in block 5 compute three address values from previously defined base and offset values. $Addr_a$ depends on a base value that has either the value Y or X, depending on whether the condition $Condition_1$ evaluates to true or false, while the offset value is fixed at 10 for all loop iterations. $Addr_b$ uses a base value that is fixed for all loop iterations and an offset that is incremented by either 1 or 2, depending on the outcome of $Condition_1$. Finally, $Addr_c$

depends on base and offset values, both of which are different for the two paths within the loop.

Let us now consider the prediction of these values, assuming the *true, false, false* sequence of outcomes for $Condition_1$ repeats itself. The conventional last value and stride predictors would not work in this case. It is possible to perform prediction using the FCM or the hybrid predictor by maintaining the patterns of values. However, new patterns of values would be generated often and the number of patterns generated would be large. Thus, frequent mispredictions can be expected.

It is possible to predict these values using control flow information. Consider the sequence of values computed by instruction 8 as shown in the figure. We can predict the next value by looking at the most recent outcome of $Condition_1$. If $Condition_1$ evaluates to false/true, the next value will be the last value of instruction 8 along the false/true branch (i.e., *Y+10/X+10*).

Consider the sequence of values computed by instruction 9. We notice that this sequence is similar to a stride sequence. In case the true branch is taken, the stride is 2; otherwise it is 1. Thus, the next value can be predicted from the stride pattern of the offset occurring along a path.

Finally, the computed value of instruction 10 depends on base and offset values, both of which depend on the control path taken from block 2 to block 5. In this case, it is possible to perform prediction by remembering both the last value and the stride values for each incoming path.

In all the above cases, we can perform the prediction for any sequence of branch outcomes. Values of an instruction along different paths are stored separately and branch history is used to predict the value on the current path. In this example, a branch history composed of the last two branches is sufficient for correct prediction. In general, outcomes from several branches may be required.

The following two sections describe prediction techniques that combine use of branch history with last value and stride predictors.

4.1 Path-based Last Value Prediction

The first scheme, Path-based Last Value (PLV) Prediction, extends the last value predictor by storing the most recent values of an instruction for different branch histories. The history of recent branch outcomes is maintained and used, along with the instruction address, to predict the next value of the instruction. For the example in Figure 4.1, two different values of $Addr_a$ are stored and then, based on the path followed, the appropriate value computed on that path is predicted.

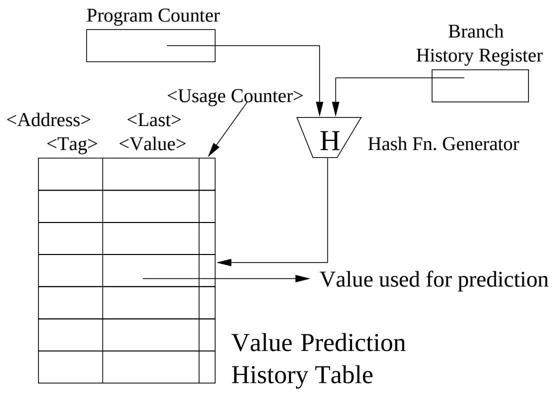


Figure 4.2: Per-path Last value Predictor Microarchitecture

Path-based last value prediction was analyzed by implementing it as a predictor whose diagram is shown in Figure 4.2. This analysis was done to determine whether it was useful to incorporate this technique within the prediction schemes of the value prediction framework. The components of the predictor that was built for the analysis are as follows:

Branch History Register (BHR): This register stores the most recent branch outcomes. After each branch is evaluated, the entire history contents are shifted left by one bit and the least significant bit is set/reset depending on whether the branch was taken/not taken. The number of paths that can be represented is limited by the size of the register.

Value Prediction History Table (VPHT): This table stores the values of instructions for different branch histories. It differs from the value prediction table proposed in [48] in that this table may store several values computed by an instruction for different branch histories. The table is set-associative and accessed using the instruction address and BHR bits. Each entry stores the predicted value and a usage counter for performing replacement. The counter is a saturating counter incremented after every correct prediction and decremented upon a misprediction. Whenever an entry needs to be replaced, the one with the lowest counter value is selected for replacement.

Hash Function Generator : The VPHT is accessed using both the address of the instruction and the history register. This unit performs the hashing of the two for indexing into the VPHT. For instruction i and BHR's value b, the function used is

$$f_1(i,b) = ((Addr(i) << Size(BHR)) + b) \bmod Size(VPHT)$$

where *Addr(i)* denotes the address value of instruction i, *Size(BHR)* and *Size(VPHT)* give the total number of distinct entries in buffer BHR and VPHT respectively. This function maps an instruction within a segment of the VPHT of size $2^{Size(BHR)}$. Within each segment, different branch histories for the same instruction are mapped to different entries of the table. This function is easy to implement in hardware and all experimental results presented in this paper are based upon this function.

For each instruction, the predictor computes the index in the VPHT by hashing the instruction's address bits with the current branch history contents. If the index maps to a valid entry, the corresponding value is used for prediction. After execution, the computed value is compared with the predicted one. Upon a misprediction, the instruction using the predicted value is squashed and re-issued with the correct value. Upon the instruction's completion, its last value information for the current branch history is updated in the VPHT.

The performance of the PLV predictor was evaluated using trace-driven simulations and instrumenting the code to perform the prediction. The design of value prediction schemes was being performed prior to implementing the prediction framework components. The main issue of interest was the prediction accuracies of the designed schemes and the type of the underlying processor being simulated was not important. The experimental framework used, consisted of an instruction-set simulator SHADE [16], a code instrumentation tool from SUN Microsystems which simulates the SPARC (Versions 8 and 9) instruction sets. The instructions of the benchmarks were instrumented to record the instruction values and branch histories. Performance was evaluated by executing the traces generated by the instrumented code over the PLV predictor's simulated microarchitecture on a 64-bit SPARC Ultra-2 processor running SunOS 5.5.1. The benchmarks were compiled using the SUN Workshop C Compiler, Version 4.2 with the -O option, and run until completion. The data input to the benchmarks is described in Table 1. Only the integer instructions were analyzed in the benchmarks. Since previous work [48] shows that floating point instructions exhibit value prediction patterns similar to those of integer instructions, similar performance is expected for floating point instructions.

The PLV predictor was implemented and its performance was compared with the performance of the last value predictor and the FCM predictor. The FCM predictor was implemented as a two-level table, with tables of both levels having the same size. The first level table entries consisted of an address tag and partial values from the last three instances of the instruction. The address tag along with the stored values are hashed to a second table storing predicted values for the instruction. The hashing function performs exclusive-OR on the most significant bits of the stored values.

For each scheme, experiments for different table sizes were performed. The different sizes of the VPHT in PLV predictor are shown in Figure 4.3. Each entry is 4-way

Benchmark	**Description**	**Input set**
Unix Utilities		
diff	File difference utility	4K C files
gawk	Pattern scanning language	parse 250K input
grep	File search utility	search 1.28M file
SPECint95 programs		
li	Lisp interpreter	train.lsp
perl	Shell interpreter	primes.in
ijpeg	Image compression	vigo.ppm
go	Chess player	2stone9.in
SPECfp95 programs		
applu	Fluid dynamics matrix solver	applu.in
fpppp	Multi-electron derivatives	natoms.in
swim	Finite difference approximations	swim.in
wave5	Maxwell's equations solver	wave5.in

Table 4.1: Description of Benchmarks evaluating Prediction Schemes

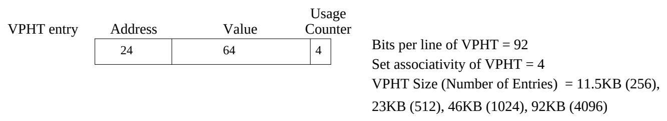

VPHT entry
Address
Value
Usage Counter
24
64
4
Bits per line of VPHT = 92
Set associativity of VPHT = 4
VPHT Size (Number of Entries) = 11.5KB (256),
23KB (512), 46KB (1024), 92KB (4096)

Figure 4.3: Table entry line for PLV Predictor

set associative. Since it is possible that some of the programs require only a small branch history and others require greater path information, experiments for three different branch history depths were performed. The branch histories used were last 2, last 4, and last 8 branches. It was observed that for smaller table sizes a small branch history performed best since larger branch histories result in significant aliasing. On the other hand, larger branch histories performed better for larger tables. Since all integer instructions were being predicted, aliasing was quite significant and in order to reduce its effects, a branch history size of 2 was selected for the experiments shown here.

Performance comparison of the PLV predictor with the last value and the FCM predictors for same number of entries per table, is shown in Figure 4.4. The figure shows the percentage of analyzed instructions that were successfully predicted. The first bar for each benchmark shows the result of applying the FCM technique. The second and third bars show results of applying the last value technique and PLV prediction technique respectively. From the comparisons, it can be seen that the FCM method did not perform as well as the other techniques for realistic table sizes. This is due to the large number of patterns that need to be stored, resulting in frequent collisions within the tables. Also, the PLV predictor outperforms the conventional predictors for most of the cases. For a 256 entry VPHT, there

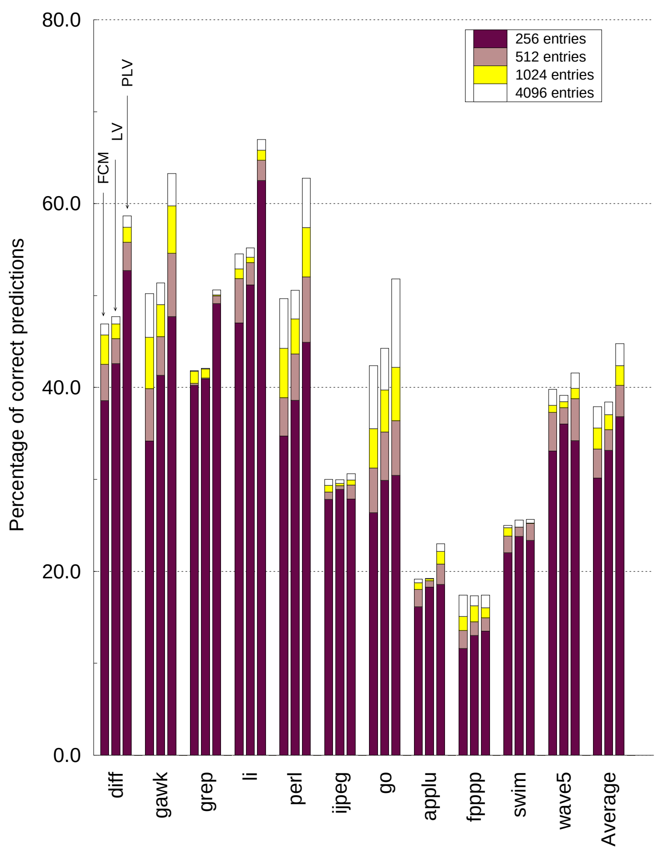


Figure 4.4: Performance of FCM vs. Last value vs. PLV Predictors in Percentage Predictions

is slight degradation in performance in cases of *ijpeg*, *swim* and *wave5*, as compared to last value prediction. The degradation can be attributed to the aliasing problem, apparent for smaller table sizes, since we need to store several values for different branch histories for an instruction. With an increase in the number of entries, the aliasing is reduced and a considerable performance improvement is observed. The average performance improvement over last value prediction is 6.4% for the table size of 4096 entries. For larger table sizes, this improvement was higher for larger branch history but this is not apparent from the results shown which use a fixed branch history size throughout.

4.2 Per-Path Stride Prediction

Two prediction schemes were also implemented to improve stride prediction by incorporating path information. The first of these two schemes, Per-Path Stride (PS) prediction, stores different stride values for an instruction along different paths. The extension of stride prediction is analogous to extending last value prediction within the PLV predictor. Path information can be useful for stride prediction when a loop variable is updated by different amounts along different paths within a loop. In the example of Figure 4.1, the variable $Offset_v$ demonstrates this characteristic. While the strides are stored separately for different branch histories, the last value is stored globally for all the strides. The table storing the strides is accessed using the instruction address and branch history bits.

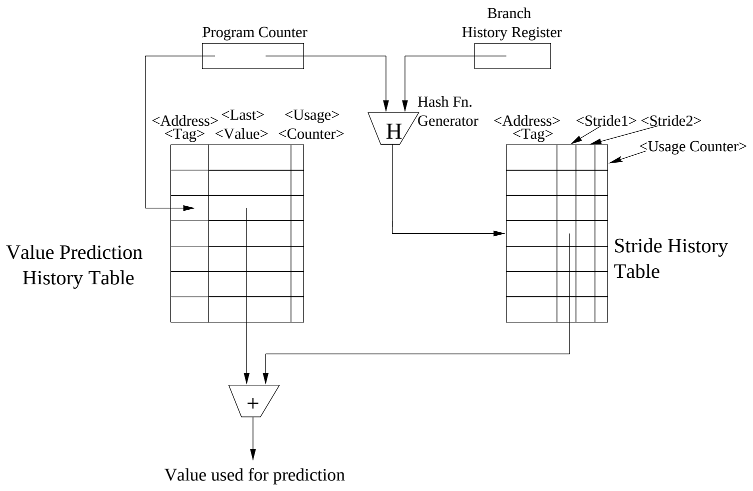


Figure 4.5: Per-path Stride Predictor Microarchitecture

Per-Path stride prediction was evaluated by implementing the scheme as shown in Figure 4.2. Besides using the components of the PLV predictor, this scheme makes use of an additional table, the **Stride History Table(SHT)**. The SHT stores the stride values for different execution histories of an instruction. It is accessed by hashing the address bits and the branch history register value. The stride update policy used here is the **two-delta policy** [21]. In this policy, the stride gets updated only if the difference between the two most recent values of an instruction occurs twice in a row. To implement this policy, two strides are stored for an instruction. The first one stores the difference between the last two values computed by the instruction while the other stride stores the value being used for prediction. The latter value is updated when the first stride has the same value twice in a row.

For each analyzed instruction, the predictor uses its lower (k bits) to map to the VPHT. The last value of the instruction is obtained from this table. Simultaneously, the address bits are hashed with the branch history to map to the SHT. This mapping accesses the stride used for the instruction and current branch history. The sum of the two values is used to predict the next value. Similar to the PLV scheme, the correctness of the prediction is checked after execution of the instruction is completed, and the update of the last value and stride is performed following completion.

The second path-based stride predictor scheme combines the previous two proposed schemes by maintaining a last value for each path, as well as a stride value for each path. This scheme, Per-Path Stride Per-Path Last Value (PS-PLV) prediction, is used to handle conditions such as that of $Addr_c$ in the example of Figure 4.1, whose value depends on operands evaluated differently along different paths. The implementation is shown in Figure 4.6. It is similar to the PLV predictor with the exception that each entry in the VPHT now also contains a stride value.

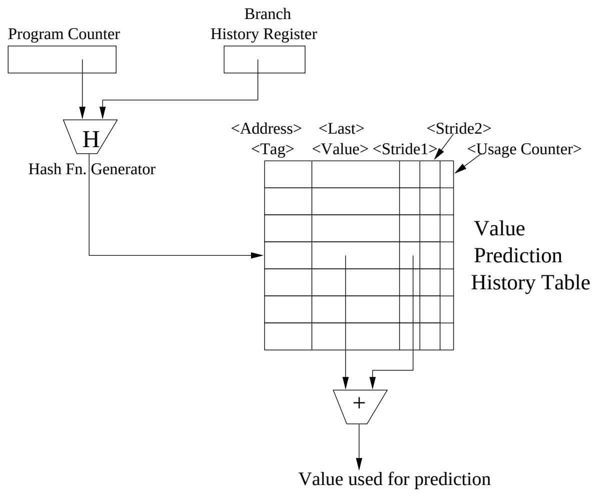


Figure 4.6: Per-path Stride Per-path Value Predictor Microarchitecture

Each instruction is hashed to index the VPHT based on its address and branch history. This table stores the last value as well as the strides for each branch history. A single access results in accessing both the values to be used for prediction. These values get updated during the completion stage of the instruction, as in the previous methods.

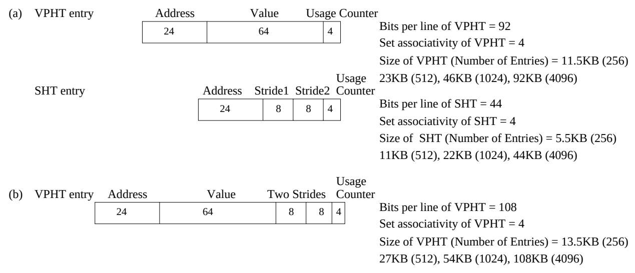

(a) VPHT entry
Address
Value
Usage Counter
24
64
4
Bits per line of VPHT = 92
Set associativity of VPHT = 4
Size of VPHT (Number of Entries) = 11.5KB (256)
23KB (512), 46KB (1024), 92KB (4096)
SHT entry
Address
Stride1
Stride2
Usage Counter
24
8
8
4
Bits per line of SHT = 44
Set associativity of SHT = 4
Size of SHT (Number of Entries) = 5.5KB (256)
11KB (512), 22KB (1024), 44KB (4096)
(b) VPHT entry
Address
Value
Two Strides
Usage Counter
24
64
8
8
4
Bits per line of VPHT = 108
Set associativity of VPHT = 4
Size of VPHT (Number of Entries) = 13.5KB (256)
27KB (512), 54KB (1024), 108KB (4096)

Figure 4.7: Table entry lines for (a) PS predictor (b) PS-PLV Predictor

The configurations of PS and PS-PLV predictors that were used are shown in Figure 4.7. They were implemented and compared with conventional stride predictors. The performances of the PS and PS-PLV predictors are shown in Figure 4.8. For each benchmark, the first bar shows the result of applying conventional stride prediction. The second and third bars show results of applying the PS and PS-PLV prediction techniques

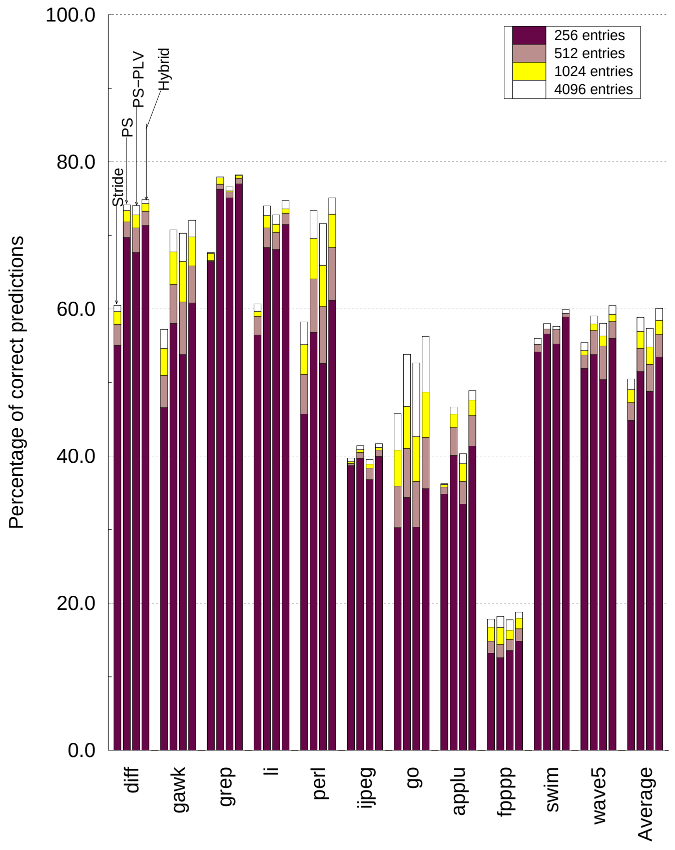


Figure 4.8: Performance of Stride vs. PS vs. PS-PLV vs. Hybrid Predictors in Percentage Predictions

respectively. The fourth bar shows the result of applying a hybrid predictor that combines the PS and PS-PLV prediction techniques. Hybrid prediction was measured for comparison with other techniques. Similar to the study for the PLV predictor, a branch history size of 2 was chosen for the analysis. From the figure, it is observed that the PS predictor almost always outperforms the conventional stride predictor (except for the benchmark *fpppp* which has a marginal degradation in performance). The PS-PLV predictor also gives a better performance when compared to the conventional stride predictor for most of the cases. However, the performance improvement is less than with PS predictor. Overall, the PS predictor shows an average improvement of 8.4% for 4096 entries while the PS-PLV predictor improves performance by an average of 6.9%.

From the above results, we infer that the PS predictor improves conventional prediction significantly. Notice that both PS and PS-PLV prediction schemes would be able to capture different sets of prediction cases (as illustrated by the example of Section 1). Hence a hybrid predictor that uses both these prediction schemes would potentially give an additive improvement in performance. Such a hybrid predictor was implemented involving both the PS and PS-PLV prediction mechanisms. The method used for prediction was based on a confidence mechanism associated with the instruction. The confidence mechanism used was a 4-bit counter value that was updated depending on which prediction mechanism performed correctly on the instruction. The performance of such a predictor is shown in Figure 4.8. The improvement in performance over conventional stride prediction for 256 entries was 10.4% and was 2% over the best path-based stride predictor. The main issue to be explored in implementing a hybrid predictor of this form is to select which predictor to use every time a prediction is to be made. In the next chapter, the predictor implemented for the prediction framework addresses this issue while incorporating hybrid prediction in the prediction framework.

4.3 Results from Experiments

From the experiments, a number of conclusions can be drawn.

(1) Path information is helpful in performing prediction of data values of instructions. Three novel schemes have been proposed for performing value prediction sensitive to path information. The first scheme, the PLV predictor, improved the last value prediction by as much as 15% for some benchmarks and 6.4% on average. The other two proposed predictors, PS and PS-PLV, extend the stride predictors. The PS prediction scheme showed an improvement exceeding 15% for some benchmarks and 8.4% on average. For PS-PLV prediction, the improvement exceeded 13% for some benchmarks and 6.9% on average. Among the stride-based schemes, PS prediction requires storage of less history (stride per path) as compared to PS-PLV prediction (both stride and last value per path). This observation favors the former scheme while implementing the predictor of the prediction framework.

(2) Value prediction displays good prediction accuracies even for small buffer sizes. In this chapter, value prediction was analyzed for all integer instructions while developing novel prediction schemes. Prediction accuracies for load instructions are even higher than general integer instructions [49]. This observation serves as a motivation to implement value prediction on the framework using the described prediction schemes. The prediction schemes were evaluated in this chapter by developing realistic predictors for each of the schemes. The value predictor in the framework need not be implemented exactly the same way as shown here. Instead, the static VLIW model can be used to optimize the design of the predictor. For example, the concept of usefulness can be used to filter the predictions, allowing a cost-efficient hybrid configuration to be incorporated. The design of the value predictor incorporates the prediction schemes described in this chapter and details of this design are elaborated in the next chapter.

Chapter 5

Value Prediction Framework Implementation

The prediction schemes designed in the previous chapter are used for implementing the framework components of a value prediction optimization. The implementation of each component and empirical results that evaluate the efficacy of the designed components are presented in this chapter. An overall analysis of value prediction on performance is also presented.

5.1 Usefulness Analyzer

In the value prediction optimization, the critical path of a region is used as a measure of usefulness in the prediction framework. In the domain of superscalar processors, the critical path is dynamic since it depends on the run-time reordering of instructions. This dynamic path may vary for different instances of the same instructions' execution. In the VLIW domain, the critical path is fixed, and thus the compiler can accurately identify the critical path and analyze the impact of predictions on the path length. Instructions whose predictions do not improve the critical path of a region need not be considered for prediction. However there are other issues in estimating usefulness using this approach that appear due to the static scheduling model. One of the challenges is that the usefulness of a prediction may not be computable independently of other predictions in the region. In some regions, several instructions may need to be predicted to improve the schedule. This situation can occur if an instruction that depends on two instructions can be speculated only if both instructions are predicted. Such a case is shown in Figure 5.1. As mentioned before, due to their high latencies only load instructions are considered candidates for value prediction in this dissertation. In this figure, the multiply instruction computing r_5 cannot be speculated unless both the loads of r_2 and r_3 are predicted. Hence the usefulness of an instruction may depend on the prediction of other instructions in the region. Another factor in VLIW prediction is that for some instructions, their prediction may not improve the instruction schedule. In Figure 5.1, predicting load r_1 will not improve the instruction schedule since the critical path of the block is dominated by loads of r_2 and r_3.

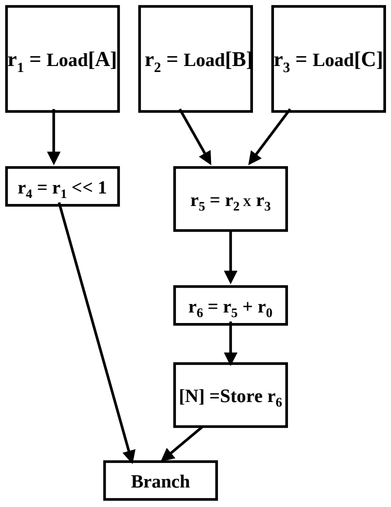


Figure 5.1: Example graph used for Usefulness Analysis

The usefulness of predictions was evaluated in the prediction framework by estimating the effect of a prediction on the the critical path length of the superblock region incorporating the prediction. Load instructions within frequently executed (hot[1]) superblock regions were analyzed. Since the compiler would have already computed dependency information for other optimizations, this phase did not involve any expensive analysis. The critical path of a region was computed using this dataflow information and the next section describes details of this computation.

5.1.1 Computing the critical path

The critical path of a region is computable by identifying the earliest time slots available to each instruction for scheduling. A *dependency graph* is used for computing the critical path, and this graph consists of all the instructions of the region as nodes and data dependencies in the form of arcs between the nodes. Each instruction node within the dependency graph of a region is associated with an *earliest starting time*, which is the earliest time slot when the instruction can be scheduled honoring all the data dependencies. On a particular VLIW machine model, due to finite resources, the actual scheduling time may be later than this earliest starting time. The earliest starting time attribute is also used by the acyclic instruction scheduler to assign priorities to the instructions.

[1]Hot regions are identified by path profiling the program and are also used for other compiler optimizations, such as superblock scheduling.

Earliest starting time for instruction P is defined to be

Estart(P) = $\text{Max}_{Q \in Pred(P)}$ (Estart(Q) + Latency(Q,P))

where Pred(P) is the set of immediate predecessors of P and Latency(Q,P) is the latency of the edge between Q and P.

Each scheduling region can contain multiple exit operations. These exit operations are branch instructions that transfer control flow to outside the region. The outgoing edges of each exit operation comprise the set of exit edges of the operation. The critical path length of a region is computed using a weighted sum of the earliest starting times of each exit operation of the region. The weighted sum is computed using probabilities that the exit operations are taken. These probabilities are obtained from edge profile frequencies.

Hence, the critical path length for a region r with exit operations $E_1, E_2 ... E_n$ is

Critical_Path_Length(r) = $\sum_{i=1}^{n}$ Estart(E_i) $*$ ($\sum_{e \in exitedges(E_i)}$ Prob(e)), where exitedges(E_i) is the set of exit edges for operation E_i and Prob(e) is the probability that exit edge e is taken.

5.1.2 Usefulness Algorithm for Value Prediction

The algorithm for identifying useful predictions is shown in Figure 5.2. The usefulness algorithm works by initially predicting all the candidate instructions for prediction (which are the loads). Next, each candidate instruction is analyzed by undoing its prediction and measuring the effect on the critical path length of the region. If the critical path length of the region increases by undoing a prediction, the corresponding load is termed as a useful prediction. In case the critical path is unaffected by undoing the prediction, or if its length reduces, the corresponding load instruction is not selected for prediction. In this latter case, incorporating the load for prediction proved to be worse for the critical path and hence the load is not considered for prediction.

As an example, consider the effects of applying the usefulness algorithm on the dependency graph of Figure 5.1. Initially all the loads are predicted, modifying the graph to the one shown in Figure 5.3(a). Each load's prediction is associated with a verification that takes one cycle (details of the verification process will be elaborated in Section 5.4). Notice that the critical path length of the region depends on the exit branch shown. Next, each predicted load is analyzed by undoing its associated prediction and observing its effect on the critical path length. Initially, the load of r_1 is considered and its prediction is undone as shown in Figure 5.3(b). Notice that undoing prediction of r_1 does not change the critical path length of the region as compared to when all loads were predicted. Hence the load is marked as not being useful for prediction. The prediction of r_2 is analyzed in Figure 5.3(c),

```
for i = 1 to num_of_loads_in_region(r)
    set $load_i$ as predicted and modify
    dependency graph accordingly;
end for;
set critical_len = critical path length of
    region r;
for i = 1 to num_of_loads_in_region(r)
    undo prediction of $load_i$ and update
    dependency graph
    set modif_critical_len = critical path
    length of updated graph;
    set diff =modif_critical_len - critical_len;
    if diff > 0
        set $load_i$ as useful;
        redo prediction of $load_i$
    else
        set $load_i$ as useless;
        set critical_len = modif_critical_len;
end for;
```

Figure 5.2: Usefulness Algorithm

and the critical path increases as a result of removing the prediction. This load instruction is useful and its prediction is kept before analyzing subsequent loads. Figure 5.3(d) analyzes load r_3 and finds it to be useful as well. Hence, the load instructions r_2 and r_3 are identified as useful predictions from the dependency graph of Figure 5.1.

The usefulness algorithm uses a backward approach by starting with all loads being predicted and then filtering the predictions that do not seem profitable. The advantage of doing this backward approach is that the complexity of the algorithm is O(n) where n is the number of load instructions of the region. Another approach is to start with no predictions and then test each load instruction for usefulness. However, this forward approach is more complex because usefulness of an instruction may not be apparent in a single pass of the instructions. An instruction that does not seem useful for prediction at first may display usefulness after predicting another instruction. For example, from Figure 5.1, the load r_2 would not be useful unless load r_3 is also predicted. Hence, the forward approach would

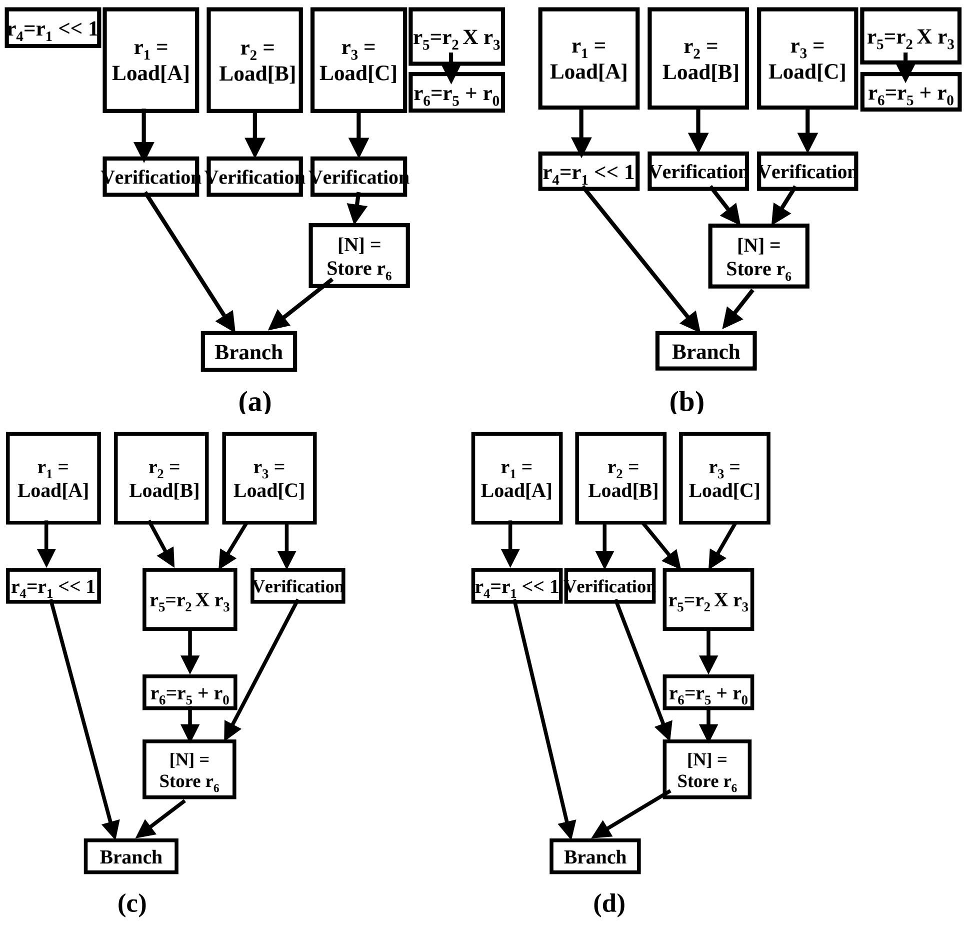


Figure 5.3: (a) Predict all loads (b) Undo prediction of r_1 (c) Undo prediction of r_2 (b) Undo prediction of r_3

require an iterative testing of the instructions for determining useful predictions. This additional level of iteration would increase the complexity of the algorithm to $O(n^2)$ here n is the number of load instructions of the region. On the other hand, the algorithm shown in Figure 5.2 is simpler while making the same computations.

Before presenting results of usefulness analyzer, the VLIW compiler and architecture infrastructure used for implementing the prediction framework are described.

5.1.3 Experimental Infrastructure

The evaluation of the value prediction optimization was performed on the *Trimaran Research Infrastructure* [1]. This infrastructure provides a vehicle for experimentation for research in compiler techniques for instruction-level parallelism. It is comprised of the following components:

- machine description language for describing ILP architectures,

- parameterized VLIW Architecture called PlayDoh [41],
- compiler front-end for C performing parsing, type checking, and a large suite of high-level optimizations,
- compiler back-end performing low-level optimizations such as instruction scheduling, register allocation and other machine-dependent optimizations,
- an extensible intermediate program representation, and
- a cycle-level simulator providing run-time information on execution time, branch frequencies, and resource utilization.

A view of the infrastructure is shown in Figure 5.4. The front end of the compiler generates the program intermediate representation(Lcode IR) that is modified into another form (Elcor IR), which is more suitable for use by the back-end. The back-end uses a description of the machine model for machine-dependent optimizations. Also, the execution statistics generated during simulation can be used for profile-driven optimizations as well as to validate new optimizations.

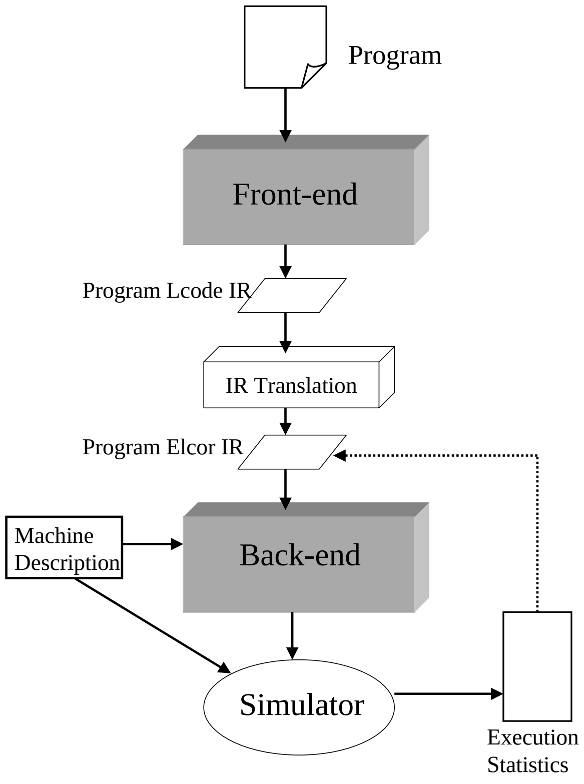


Figure 5.4: Trimaran Infrastructure

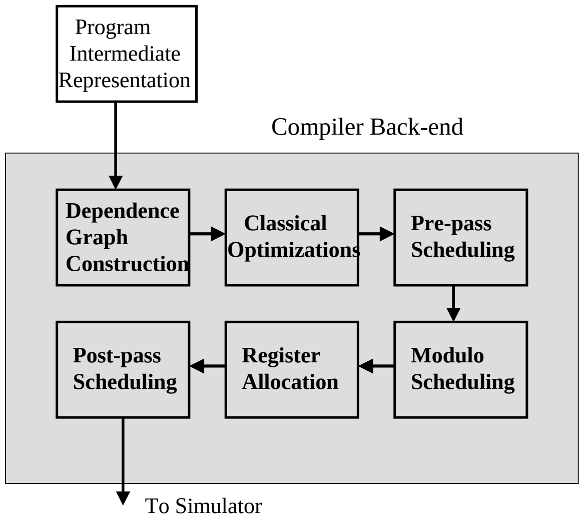


Figure 5.5: Trimaran Compiler back-end

The prediction framework developed in this dissertation exploits parallelism at the instruction-level. Hence, the implementation of value prediction was applied in the back-end of the compiler performing fine-grain optimizations. A pictorial view of the compiler back-end is shown in Figure 5.5. The back-end comprises of a suite of classical optimizations that include constant folding, common subexpression elimination and dead code elimination. Although these optimizations were applied in the front-end, they are re-applied because of additional redundancies that surface due to the translation of the intermediate representation. The back-end also consists of two instruction schedulers (pre-pass and post-pass), a register allocator and a modulo scheduler for the loops. Instruction scheduling and register allocation are performed on a per-region basis. The compiler infrastructure has the capability to form regions including Superblocks [40] and Hyperblocks [50]. A superblock is a block of instructions in which control enters from the top but may leave at one or more exit points. Superblocks enable the compiler to optimize beyond basic block boundaries by removing the constraints due to side effects within a sequence of blocks. These constraints are removed by selecting a trace of blocks using profile information and performing tail duplication [40] to eliminate side entrances to the trace. A hyperblock is a similar structure to the superblock with the difference that instructions within the hyperblock are predicated. In case of hyperblocks, basic blocks from different control flow paths are grouped together into a single extended block. This grouping can result in a slow-down of the included paths if each path requires almost all the available resources resulting in resource dependencies in the hyperblock. Also, the increase in code-size is larger than in the superblock case since tail duplication is applied across multiple paths. Although techniques such as partial-reverse if-conversion [77] have been proposed to overcome these factors, they are not implemented in Trimaran. This dissertation uses superblocks for region-based optimizations due to their consistently good performance.

5.1.4 Benchmarks used for evaluation

The Trimaran Infrastructure is currently supported by the benchmarks within SPECint95 and Mediabench [44]. The benchmarks used for the analysis of the prediction framework are shown in Table 5.1. The benchmarks from the Mediabench were included because of increasing presence of multimedia applications in general computing and since several embedded processors implement VLIW machine models. Among the Mediabench benchmarks, video (*mpeg*), image(*epic*) and speech (*rasta*) benchmarks were used. Among the SPECint95 suite, the benchmarks that passed all the compiler phases including superblock optimizations were incorporated. The SPECint95 benchmarks that were not considered were the ones that ran out of virtual memory during compilation on the underlying processor (a 733 Mhz dual processor machine with 1G virtual memory). All of the analyzed benchmarks were run until completion. The profiler ran using one set of inputs and the performance was analyzed using a different input set. For the SPECint95 benchmarks, the *train* and *reference* inputs provided with the benchmark suite were used. The Mediabench benchmarks used the two data sets provided with the benchmarks for training and analysis.

Benchmark	**Description**
Mediabench programs	
epic	Image data compression
rasta	Speech processing
mpegdec	Video decoder
SPECint95 programs	
compress	Compression utility
li	Lisp interpreter
ijpeg	Image compression
m88ksim	Chip Simulator
vortex	Object-oriented database

Table 5.1: Description of Benchmarks

5.1.5 Usefulness Analysis Results

The usefulness analyzer was implemented on the above experimental infrastructure and Table 5.2 shows the fraction of predictions reduced due to the analysis. The Table shows three columns for each benchmark. The first column indicates the total candidate predictions. This number indicates the number of loads predicted if no usefulness analysis was applied. This number does not include loads due to register spills since value prediction is performed in the compiler before register allocation. The second column for each benchmark indicates the number of loads predicted after applying the above algorithm. In both cases, loads were predicted only if their prediction accuracy exceeded a threshold. This estimation of predictability for an instruction was done by the profiler and is explained in the next section. From the Table, it is observed that usefulness reduced the predicted loads by approximately 85%. The third column shows this percentage reduction for each benchmark.

Benchmark	*Total candidate loads*	*Useful loads*	*Percentage reduction in predictions*
epic	45	10	77.8
rasta	424	64	84.9
compress	306	45	85.2
mpeg2dec	443	103	76.7
li	487	68	86
m88ksim	1176	172	85.3
ijpeg	1989	307	84.5
vortex	2368	391	83.5

Table 5.2: Usefulness Analysis

The key features of the usefulness analyzer are

- a simple heuristic-based O(n) algorithm was able to effectively identify useful predictions.
- usefulness information reduced the predictions by 85% on average.

5.2 Selective Profiling

The instructions deemed useful by the usefulness analyzer were profiled for prediction accuracy. Several prediction schemes were used to analyze the value traces to test the value predictability of the useful loads. The prediction schemes used for this analysis include last value [49], stride [29], context-based [64] along with the path-based schemes developed in this dissertation and described in Chapter 4. Among the context-based schemes, the finite context method (FCM) with R-5 context function [63] was used. Amongst the path-based schemes, the per-path stride prediction and per-path last value schemes were used because of their higher prediction accuracies and lower hardware requirements. Besides individual schemes, the efficacy of hybrid prediction was also analyzed. Previous work related to hybrid predictor design has focused on implementing at most three individual prediction schemes and selecting the best scheme using saturating counters [76]. The disadvantage of implementing more than three schemes is that the learning time of an instruction towards a specific prediction scheme increases. The hybrid configurations analyzed in this dissertation involved three prediction schemes, namely last value, stride and a two-level prediction scheme. The profiler analyzed three different hybrid configurations (1) hybrid of last value, stride and FCM, (2) hybrid of last value, stride and PLV and (3) hybrid of last value, stride and PS. Last value and stride prediction were always included because of their simple implementation. For each instruction, the prediction accuracies for individual and hybrid schemes were computed. Any instruction with the best prediction accuracy higher than a fixed threshold was deemed as being predictable.

Before evaluating the overhead due to selective profiling, the tools available for profiling within the experimental infrastructure are described below.

5.2.1 Profiling within Trimaran

The Trimaran Compiler Infrastructure consists of two kinds of profiling tools.

- *Front-end profiler* : This profiling tool instruments the program intermediate code produced by the front-end of the compiler. The profiler is used to generate control flow profiles to apply for function inlining and region formation

- *Performance monitoring tool in simulator*: This tool collects accurate program statistics information and execution information generated during program execution. A profiler is provided within the simulator and consists of the following two components:

    - Lightweight profiler: This component analyzes the control flow behavior while the program is being executed on the simulator. The statistics evaluated by this component include cycle counts, resource utilization and dynamic operation count.

    - Trace-driven profiler: This component provides a larger range of information to the compiler such as memory addresses accessed and cache behavior. Such information is generated as a trace during simulation.

The performance monitoring tool is slower than the front-end profiler because the former modifies the execution functionality of the simulator to collect raw data and processes it into profile information. However, the generated information is more accurate than the data emitted by the front-end profiler. In the latter case, the profile information can become inaccurate by optimizations within the compiler back-end. Within the performance monitoring tool, the trace-driven profiler is slower but produces more detailed execution information than the lightweight profiler. Among these profiling tools, this dissertation uses different profilers in different contexts. As shall be seen throughout the framework implementation, the most appropriate profiler tool is chosen depending upon the amount and accuracy of profile information required.

5.2.2 Profile Results

When selective profiling was implemented for value prediction in the front-end profiler, the profile data did not correlate to execution statistics, even on the same program inputs. The reason, as mentioned above, was that optimizations in the back-end of the compiler contaminated the profile information. For instance, control speculation hoisted instructions to execute the instructions a greater number of times than when they were profiled. An instruction found to generate predictable values after profiling in the front-end may get control-speculated above a branch during instruction scheduling, thereby increasing its execution frequency. It is possible that the additional execution instances of the instruction generate unpredictable values (corresponding to error conditions handled by the branch above which the instruction was speculated). In such cases, if speculation was to be turned off, control would not have reached the instruction by taking the branch. However, turning off speculation would result in conservative instruction schedules.

Hence, to maintain high accuracy of profiles by profiling as late as possible, the performance monitoring tool in the simulator was used to profile programs. Although profiling each instruction was slower in the simulator than instrumenting in the front-end profiler, one advantage was that the program size was not affected by additional code. This effect is beneficial in terms of cache performance. Since instruction-level profile information was required, the trace-driven profiler of the simulator was extended to generate values using a tracer. Only the load instructions termed as being useful by the usefulness analyzer generated values in the trace. The prediction schemes discussed in the beginning of Section 5.2 were incorporated to analyze the values generated by the tracer. Both the trace generation and analysis formed part of the implementation of the selective profiler and were performed as the program executed on train inputs. Hence the overhead of selective profiling was incorporated within the program execution time. For profile accuracy, instructions were not aliased with each other for all the schemes except FCM prediction. FCM predictors use a two-level scheme [63] where the first-level table is accessed by the instruction's address, and the history of pattern values stored in this table are used to access the second-level table. In case of the second-level table, since an exponential number of patterns would be generated, each pattern could not be allocated an individual table entry without running out of memory. Instead a very large table (8M entries) was used to reduce aliasing. Figure 5.6 shows the percentage reduction in execution overhead due to selective profiling. The first bar for each benchmark shows execution overhead when all the candidate load instructions were profiled. The second bar shows the overhead when only the useful loads were profiled. From the figure, we see that the execution overhead was substantially reduced due to selective profiling. Note that the percentage reduction in execution overhead correlates to the reduction in the number of profiled loads from Table 5.2. The exception is the case of *ijpeg* where percentage reduction in execution overhead far exceeds the reduction observed for any other program. In this case, the execution of the loads not found to be useful is a significant part of the execution of the program. By avoiding these loads, the profiling overhead is substantially reduced.

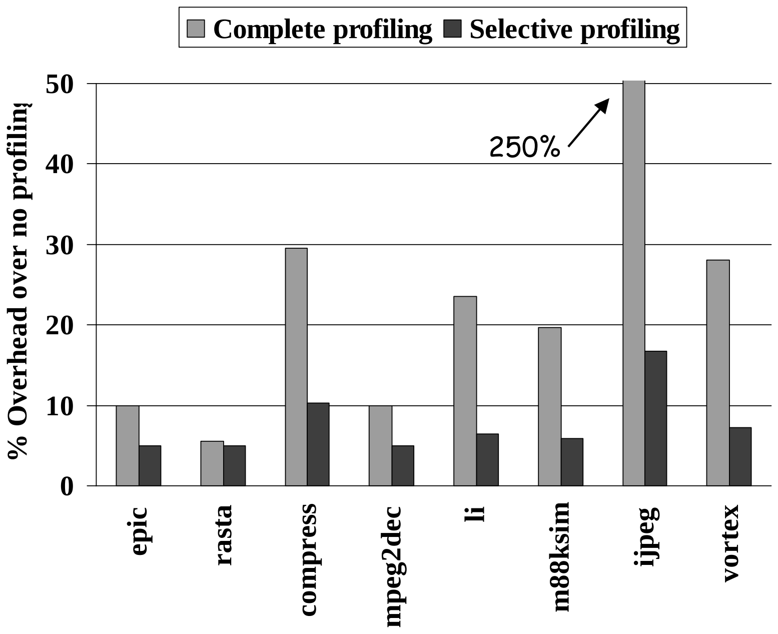


Figure 5.6: Percentage execution overhead due to Value Profiling

The key features of the selective profiler were that

- trace-level simulation was used to analyze the values of instructions,
- individual prediction schemes as well as hybrid configurations were analyzed for prediction accuracy, and

- incorporating usefulness information reduced the overhead of profiling substantially.

This framework component utilized the prediction schemes in its implementation. The next section shows how these schemes were incorporated to perform run-time prediction of values.

5.3 Predictor

The predictor component of the framework is designed in hardware to store the run-time history for making predictions on values. The selective profiler used several different prediction schemes and hybrid configurations to profile programs for prediction accuracy. The design of the predictor needs to be configured to the most effective prediction schemes and hybrid configurations. One way of doing so is to assign each instruction an entry in the predictor table and configure this entry to the most suitable prediction schemes (from profile data) for that instruction. Different instructions would be able to accurately predict values using different amount of prediction history. This approach can be applied in the static VLIW domain since the most suitable prediction schemes can be determined for instructions at compile-time. However, configuring the predictor entries for different programs would require reconfigurable hardware. Another approach is to design a hybrid predictor with multiple columns per entry to store the history for several prediction schemes. The actual prediction schemes utilizing these columns for each predictor entry can vary for different instructions (depending on the most profitable hybrid configuration for the instruction accessing that entry). The advantage here is that fixed hardware can be used and all predicted instructions are not subject to the same prediction schemes. The latter approach is used in this dissertation, and the following section describes this design and other details of the predictor design.

5.3.1 Predictor Configuration

The value predictor was designed by analyzing the characteristics of different prediction schemes and building the predictor around the most effective schemes. This design was a one-time process in the framework and all benchmark programs for this optimization used this designed predictor. The effect of individual optimizations was not important in this general study of value predictability of programs since the analyzed information was not used as profile data. Hence, the analysis was performed using the front-end profiler which was faster than simulation. Using the front-end profiler also allowed the entire SPECint95 suite to be used for building the predictor. Figure 5.7 shows a distribution of predictable instructions among the different prediction schemes. This figure shows a distribution of predictable instructions across the various prediction schemes. An instruction was deemed to be predictable if its prediction accuracy was greater than a threshold of 65%. This threshold was kept reasonably low to allow several instructions to qualify as being predictable. Further reduction in the threshold value increases the percentage of mispredictions sharply. For each scheme shown, the fraction of predictable instructions most accurately predicted using that scheme is displayed. For example, in case of *compress*, 52% of predictable instructions are best predicted using last value, 7% using stride, 33% using FCM and so on. As in the selective profiler, instructions were not aliased for all schemes except for FCM prediction with the latter using a large second-level table (8M entries).

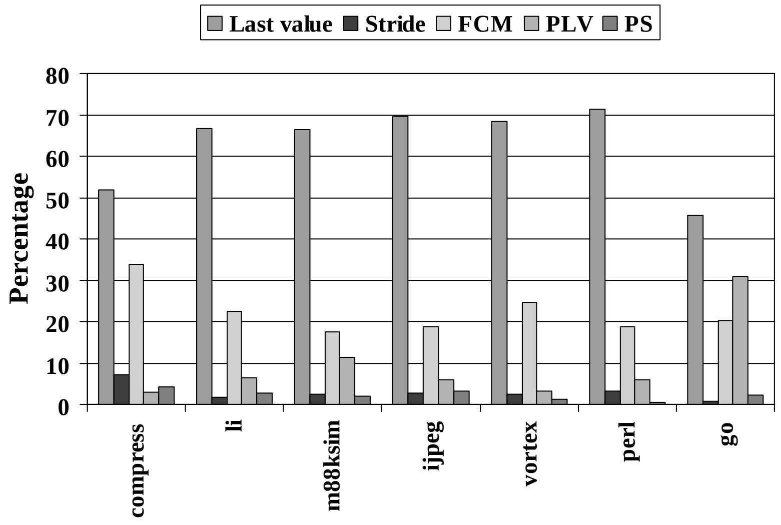


Figure 5.7: Percentage predictable instructions best predicted by each scheme

From Figure 5.7, we see that a majority of the predictable instructions were best predictable using last value prediction followed by FCM prediction. Stride and path-based predictors account for about 10-15% of the predictable instructions. From this study we can see that each of the analyzed prediction scheme is able to predict different fractions of the instructions accurately. Although last value prediction seemed to outweigh other schemes in predicting the most fraction, the other prediction schemes also need to be considered in the predictor design. It is possible that hybrid prediction may be able to capture the various prediction schemes in an effective way. In order to analyze the efficacy of hybrid prediction, another study was performed. This study computed the fraction of predictable instructions whose accuracy improved by hybrid prediction as compared to using a single prediction scheme. This comparison would give insight about how often hybrid prediction was more efficient in general than using a single prediction scheme. Each instruction was predicted

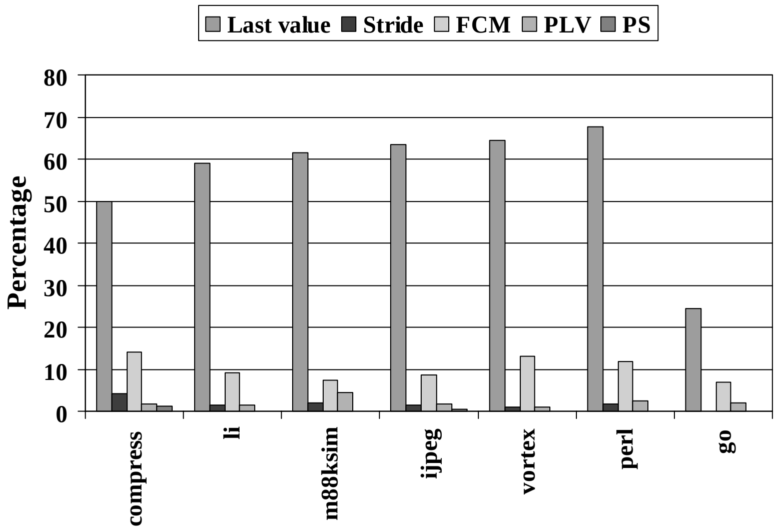


Figure 5.8: Percentage predictable instructions (categorized by best scheme) that do not improve using hybrid scheme

using the three hybrid configurations outlined in the selective profiler. Figure 5.8 shows that about 50% of predictable instructions were best predicted using last-value prediction and their prediction did not improve their prediction accuracy using any sophisticated hybrid predictors. This study indicates that the about half the predictable instructions need only a simplified predictor entry (for storing last value). The other half of the predictions require more sophisticated prediction techniques.

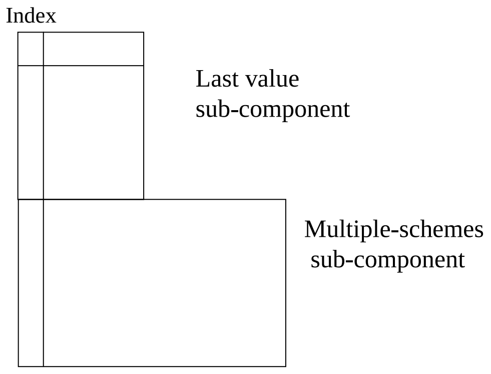


Figure 5.9: High-level view of Segmented Predictor

The instructions that are best predictable using their last value do not need any additional history. On the other hand, the instructions that improve prediction accuracy using hybrid prediction would require larger amounts of prediction history to be stored. In order to accommodate both of these cases, the predictor was designed as a segmented predictor having two partitions as shown in Figure 5.9. The first component stored the instructions that needed only their last value for prediction and the second component stored history for multiple prediction schemes. The hybrid partition is shown to have wider entries since it needs to store history of multiple prediction schemes along with confidence mechanisms. The partitions were created with equal number of entries since about half of predictable instructions were best last-value predictable, as shown in Figure 5.8. The compiler would decide from the profile data, which partition to use for predicting an instruction. This information would be annotated to the predicted instruction and used during run-time prediction to store the predicted value in the appropriate partition.

The next section describes the design of the hybrid partition in detail.

5.3.2 Hybrid partition design

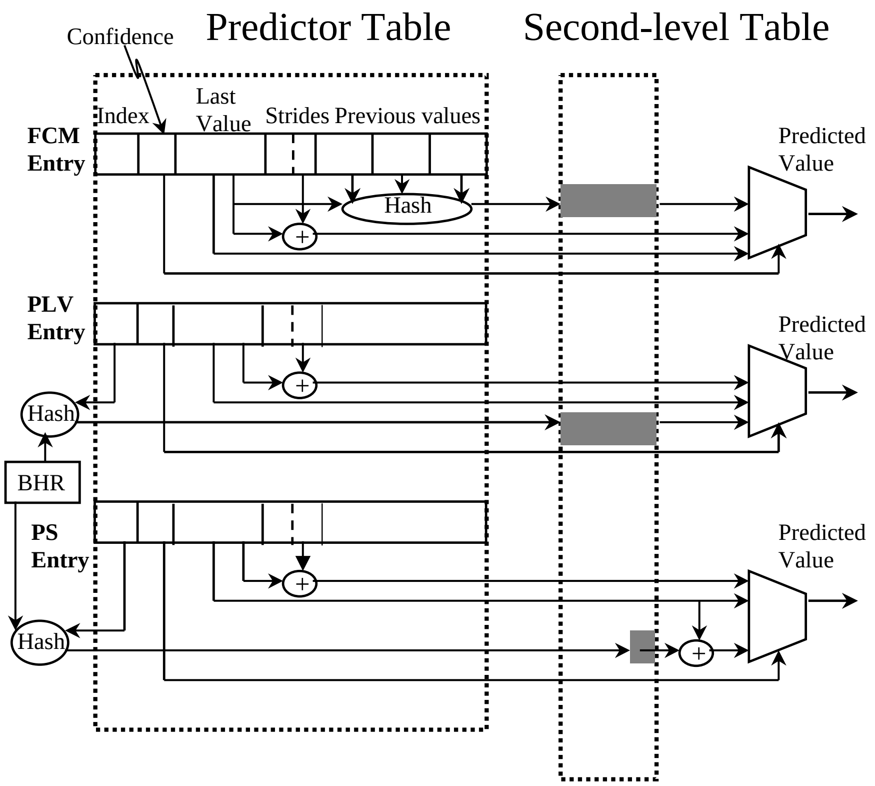


Figure 5.10: Multiple-schemes Predictor Sub-component

The hybrid predictor designed in this dissertation incorporates previous prediction schemes and the newly developed path-prediction schemes. As mentioned before, the number of prediction schemes for hybrid prediction was restricted to three and the hybrid configurations developed used last value, stride and a two-level prediction scheme. However the hybrid configuration for an instruction was based on the predictability behavior of the instruction. Since the predictions are decided statically by the compiler, it is possible to select the prediction schemes most suitable for an instruction using profile data. Hence, different hybrid schemes can be interleaved within the same underlying hardware structure. Figure 5.10 shows the design of the hybrid component of the predictor. Each predictor entry uses last value and stride prediction in the hybrid scheme. The third prediction scheme is either FCM, path-based last value or path-based stride prediction scheme. The profile data is used to determine which of the three prediction schemes is most beneficial for predicting the instruction, and the instruction is predicted accordingly. Each of these three schemes is implemented as a two-level scheme. The FCM entry is used for the instructions using context-based prediction and the previous four values are hashed together to access the second-level table. The PLV entry implements path-based last value prediction in a hybrid fashion. The predictor table index is hashed together with the branch history register contents to access the second-level table for the path-sensitive last value. Similarly the PS entry accessed the path-sensitive stride using the branch history. In case of the PS entry, only the lower 8 bits of the second-level table were used for the path-sensitive stride. Each of the three types of predictor entries uses confidence bits to select the best prediction scheme.

Each of the predictor entries uses the same entry size in the first-level table. Hence the same table entries can be used inter-changeably for the two prediction schemes (in case aliasing cannot be avoided). In case of the PLV and PS entries, part of the table entry is unused. Although it seems that the hardware is not being utilized efficiently, the number of such cases are infrequent. From Figures 5.7 and 5.8, it was observed that the hybrid component of the predictor is dominated by FCM entries and the percentage of path-based entries are about 10% on average. Another observation from Figure 5.10 is that last value and stride prediction were fixed for all entries allowing different hybrid configurations to be interleaved (based on the third component of each entry which was consistently a two-level scheme).

5.3.3 Static Predictor Assignment

One of the features of the predictor is that the instructions are statically selected for predictions. This feature can be used to assign entries of the predictor table to the instructions being predicted. This static assignment can be performed to reduce chances of aliasing among different predictions. Aliasing has been shown to be a major concern in value prediction tables, especially in limited size tables [56].

The algorithm developed for static predictor assignment is shown in Figure 5.11. The algorithm allocates predictor entries to instructions and these entry indices are stored in the instruction format. While making predictions, the entry index is read as part of

```
Assign_predictions(Table T, Predictions P)
( Each table entry t is associated with a weight
  $w_t$ = sum of profile weights of all assigned
      predictions to index t )

If Size(P) $\leq$ Size(T)
      assign each prediction in P a unique table
      entry of T
else for each prediction $p_i$ ( i = 1 .. Size(P) )
      If i $\leq$ Size(T)
         assign the $i_{th}$ table entry of T to $p_i$
         set $w_i$ = profile_weight($p_i$)
      else
         min = table entry with minimum profile
         weight $w_{min}$
         assign the entry min to $P_i$
         $w_{min} = w_{min}$ + profile_weight($p_i$)
```

Figure 5.11: Static Predictor Assignment Algorithm

the instruction. Details of the instruction format are provided in the section describing the speculation transformer. The main idea of the algorithm of Figure 5.11 is to prevent the more frequently executed predictions from overlapping. In case aliasing cannot be prevented, less frequently executed predictions are aliased with the more frequently executed predictions. Each predictor table entry is associated with a cost that is the sum of the profile weights of all the predictions assigned to that entry. For each prediction the table entry with the least cost is chosen for assignment. The complexity of the algorithm is $O(n^2)$ where n is the number of predictions of the predictor. The algorithm is executed separately for both the last value and hybrid partitions.

5.3.4 Prediction Accuracies

The above predictor design was implemented to predict program instructions. The predictor was implemented as a 256 entry table because usefulness analysis from Table 5.2 indicated that the average number of predictions were usually in that range. The entries were divided evenly into two partitions - last value and hybrid. The second level table, used by path-based and context-based prediction, was selected to be 16K entries. The second-level table was smaller than the one used in the profiler since the number of actual predictions were reduced significantly as a result of usefulness analysis and profiling. The recent four branches were used to estimate the path for path-based prediction. The prediction table was also initialized by the compiler with estimated initial predictions. The initial predictions of a program were available from its profile data and inserted in the predictor table. Initializing the predictor reduced the learning time of a prediction scheme. In case multiple prediction histories were stored for a table entry, an initialized value was inserted for each of the histories.

Table 5.3 shows the prediction accuracies of the designed predictor. The table shows three columns of prediction accuracies expressed as fractions for each benchmark. The first column gives the prediction accuracy when no usefulness information is used to filter the predictions. The second and third columns incorporate usefulness information. In the second column, the program counter is used to access the predictor table. In the third column, the static assignment algorithm is used to assign table entries to predictions. For example, for the *rasta* program, the prediction accuracy without usefulness information is 77%. When usefulness information is applied and the program counter is used for table accesses, the accuracy becomes 80%. In case of using the static assignment algorithm along with usefulness information, the prediction accuracy improves to 91%. For each benchmark, only instructions indicated as predictable by the profiler were predicted. A threshold of 65% prediction accuracy was used to assign an instruction to be predictable.

From Table 5.3, it is observed that if usefulness information is not incorporated, the prediction accuracy is low with an average of 60%. This average was lower than the prediction threshold because of aliasing effects. Incorporating usefulness information improves the accuracy for each benchmark. If static predictor assignment is used to access the predictions, the accuracy is higher by 4% on average than when the program counter is used for indexing.

The key features of the predictor component are as follows:

- The predictor table is partitioned using profile information. The smaller partitions predict instructions that use simple prediction schemes,
- hybrid prediction is achieved by interleaving different hybrid schemes over the same hardware, and
- static assignment of predictions to table entries reduces aliasing.

Benchmark	*No usefulness*	*Usefulness, No static assignment*	*Usefulness and static assignment*
epic	0.65	0.70	0.70
rasta	0.77	0.80	0.91
compress	0.55	0.66	0.67
mpeg2dec	0.58	0.78	0.78
li	0.45	0.69	0.71
m88ksim	0.82	0.93	0.95
ijpeg	0.43	0.70	0.71
vortex	0.44	0.81	0.87
Average	0.60.	0.73	0.77

Table 5.3: Prediction accuracy as a fraction

5.4 Speculation Transformer and Restorer

This section discusses the speculation transformer and restorer components of the framework. The restorer was implemented both in software (static recovery) and hardware (dynamic recovery). The functionality of the speculation transformer depends on the implementation of the restorer. Hence both these components are presented together for each of the static and dynamic case.

5.4.1 Static Recovery

Static recovery was implemented similar to previous work of [28] in which each prediction is associated with a recovery block. This previous work speculates instructions dependent on a single prediction. This dissertation extends this previous work by analyzing multiple predictions within a region. Instructions dependent on several predictions are allowed to execute speculatively, and static recovery for these cases is investigated.

Before presenting the static recovery model, the semantics of the speculation transformer are described.

5.4.1.1 Speculation Transformer

This framework component performs static code restructuring to incorporate value prediction and speculation. In order to perform value prediction, the instruction set of the VLIW architecture is extended. The extensions proposed in previous work [28] are modified in this dissertation to make them more appropriate to the prediction framework. The following operations are used by the speculation transformer. For each operation type, the operands are shown in <> and any additional annotated bits are in parenthesis.

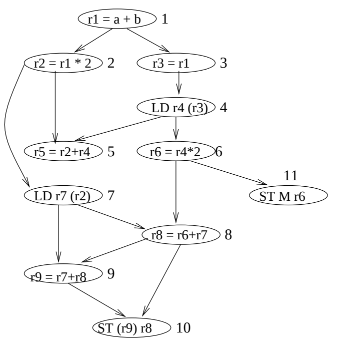
- Load Predict ($LdPred < r >< TableIndx >$): This operation reads the predicted value from the index $TableIndx$ of the predictor table. This operation is only used for the predictions accessed from the hybrid partition of the predictor. In this case, an additional cycle is required to read the predicted value because the second level table may need to be accessed for prediction. Within the hybrid partition, the predicted value is selected based on the confidence measures of the individual prediction schemes and loaded into register r. The confidence measure of the selected scheme is speculatively updated assuming the prediction to be correct. On the other hand, when the prediction is made from the last value partition, an additional cycle is not necessary to read the value. The reason is that the last value can be accessed similar to a register read within an operation. In this latter case, an explicit $LdPred$ operation is avoided and the table index is encoded within the instructions that read the predicted value for speculation. This way the speculated operation can be initiated in the first cycle of execution of its parent block. Also, the use of an additional register to store the predicted value is avoided, reducing pressure on the register allocator. This functionality is provided by allowing operations to access the predictor table similar to a register file read.

- Update Predictor ($UdPred < r >< TableIndx >$): This operation updates the predictor entry $TableIndx$ with the value r. This operation is executed during a misprediction and is used for both the last value and hybrid partitions. In case of last value prediction, the correct last value is inserted in the predictor. In case of hybrid prediction, the correct value is inserted and the confidence counters are updated for each prediction scheme.

All other operations are categorized as being speculative or non-speculative.

- Speculative operations ($< Opcode >< dest >< src_1 >< src_2 > [Spec]$): These operations utilize a predicted value either directly or indirectly through some data dependency. A speculative operation is executed similar to a generic instruction except that predicted values are obtained either directly or indirectly through a previous $LdPred$ operation. An additional bit is added to the operation format (shown as $Spec$) to specify whether the operation is speculative or not.

- Non-speculative operations ($< Opcode >< dest >< src_1 >< src_2 > [Spec]$): These operations utilize only correct (verified) values. Examples of non-speculative operations are stores and branches. Also, operations that do not depend on any prediction are termed non-speculative. These operations also have an additional bit (*Spec*) which is reset to distinguish them from the speculative operations.



(a)

Cycle	VLIW Instruction		
1	r1 = a + b		
2	r2 = r1 * 2	r3 = r1	
3		LD r4 (r3)	LD r7 (r5)
4			
5			
6	r5 = r2+r4	r6 = r4 * 2	
7	r8 = r6+r11	ST M r6	
8	r9=r11+r8		
9		ST (r9) r8	

(b)

Figure 5.12: Example (a) Dependency Graph (b) Schedule with no prediction

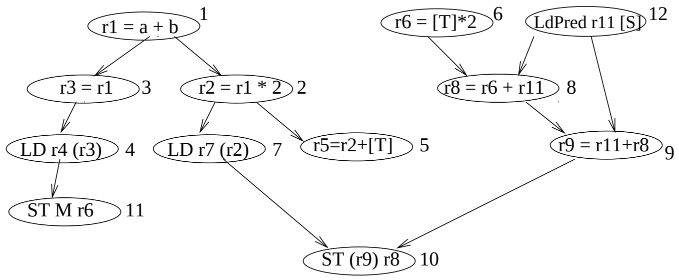


Figure 5.13: Modified Dependency Graph due to prediction

To understand the significance of these operation forms, consider an example of VLIW code scheduled on the modified machine. Figure 5.12 shows a dependence graph for a sequence of operations. Assume the add, move and multiply operations are of unit latency and the loads are of latency 3. This figure shows the schedule of the VLIW code without any value speculation. The VLIW processor supports 2 memory operations and 4 ALU operations within an instruction. For simplicity, the figure only shows relevant operations per instruction and omits details of exact issue slots within an instruction. Operations 4 and 7, being loads and of high latency, are useful for prediction since they would allow several operations dependent on them (5,6,8,9,10,11) to be speculated. Assume that the value computed by operation 4 ($r4$) is last value predictable, while the value computed by operation 7 ($r7$) is predictable using a hybrid predictor. The dependency graph modified after prediction is shown in Figure 5.13. In this figure, the prediction of $r4$ is read by operations 5 and 6 by encoding the table index for that prediction with the operations. On the other hand, since $r7$ is generated using hybrid prediction, an *LdPred* operation is used to store the predicted value into a separate register. The store operations (10 and 11) are not speculated. The verification of the predictions are described in the next section.

5.4.1.2 Static Restorer

Verification of a predicted value involves re-execution of the original operation, whose value was predicted, and a comparison with the predicted value. If the comparison indicates a misprediction, the operations that were speculated using the predicted value need to be re-executed for recovery. Static recovery in value prediction has been proposed previously by creating a recovery block for each prediction [28]. This recovery block contains the *UdPred* operation and operations speculated using the corresponding prediction. After completion of recovery, control needs to be returned to the region performing the prediction. In the context of superblock regions used in this dissertation, control cannot enter a superblock anywhere except at the top. This problem is analogous to the elimination of side entrances during superblock formation and is addressed using *tail duplication* [40]. In this process, the operations executed in the predicted block subsequent to the verification of the prediction are duplicated in the recovery block. A high-level view of a superblock incorporating a prediction and its recovery block is shown in Figure 5.14. In this figure, the shaded part of the superblock consists of the operations appearing after the verification (indicated by the $BRNEQ$ operation) of the prediction. These shaded operations are duplicated in the recovery block for execution (along with $UdPred$ and speculated operations) in case of a misprediction.

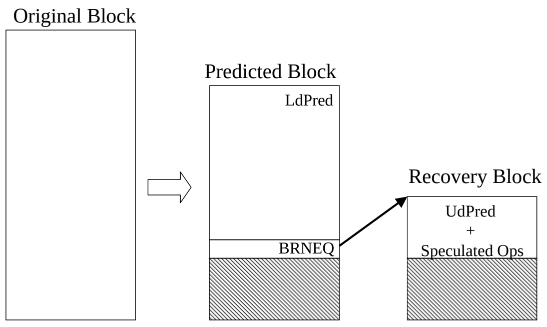


Figure 5.14: Static Recovery using Tail Duplication

The above figure shows a case of recovery for a single prediction with a superblock region. In case of multiple predictions within a superblock region, static recovery can be performed in two different ways.

- Separate recovery block per prediction: In this case, a separate copy of the speculated operations is created for each prediction. This case is shown in Figure 5.15. This figure shows two predictions being performed. The operations appearing after the first prediction check ($BRNEQ_1$) are duplicated in the recovery block 1 and operations after the second check ($BRNEQ_2$) are duplicated in the recovery blocks 1 and 2. The latter case of duplication is performed twice creating three copies of the operations (shown in lighter shading). This redundancy is higher as the number of predictions in a region is increased. Another effect of this approach is that an operation speculated based on more than one prediction would appear in multiple recovery blocks. Such an operation would be placed within the recovery block of each prediction that the operation depends upon. Hence, code growth is the main issue when each prediction is assigned a separate recovery block.

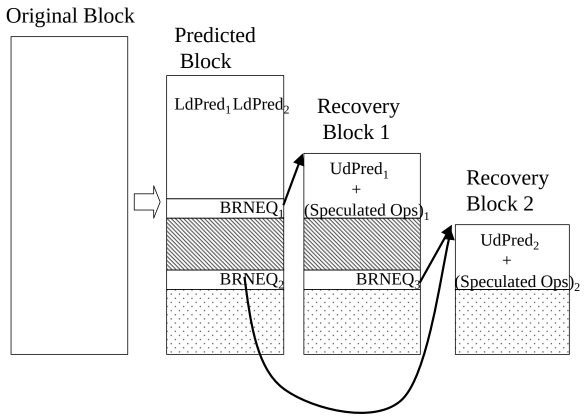


Figure 5.15: Static Recovery for multiple predictions with separate Recovery Blocks

- Single recovery block for all predictions in a region: In this case, a single block incorporates all the instructions speculated within a superblock region. Based on which predictions are incorrect, the corresponding speculated operations are re-executed from the recovery block. The rest of the operations within the recovery block are nullified using predication. The predicates are generate by the prediction checks. This approach uses a new instruction BR_OR that ORs the predicates generated by the prediction checks and branches to recovery code if the resulting predicate is true. The instruction format of this operation supports up to four predicate inputs. Figure 5.16 shows the high-level view of this approach. The predicates are generated during the verification of the predictions and ORed. Control transfers to the recovery block in case any of the predictions is incorrect (when p_3 is true). The advantage of this method is that recovery code is not duplicated multiple times as in the previous approach. Also, mis-speculated operations are re-executed only once during recovery, even if they depend on multiple predictions. An overhead to this approach is the additional operation to OR all the predicates of the prediction checks.

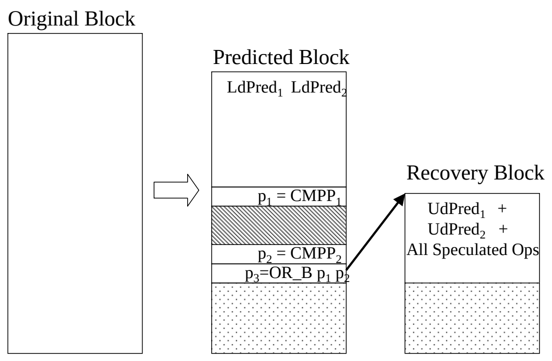
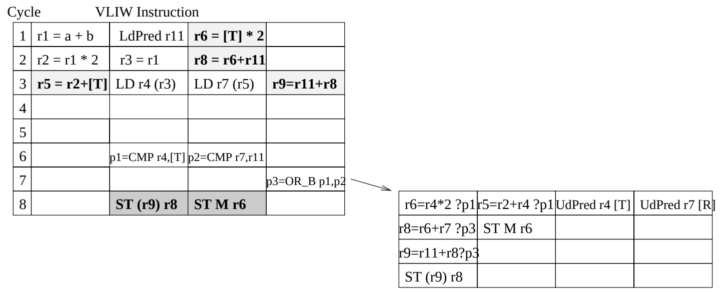


Figure 5.16: Static Recovery for multiple predictions with single Recovery Block



Cycle	VLIW Instruction			
1	r1 = a + b	LdPred r11	**r6 = [T] * 2**	
2	r2 = r1 * 2	r3 = r1	**r8 = r6+r11**	
3	**r5 = r2+[T]**	LD r4 (r3)	LD r7 (r5)	**r9=r11+r8**
4				
5				
6		p1=CMP r4,[T]	p2=CMP r7,r11	
7				p3=OR_B p1,p2
8		**ST (r9) r8**	**ST M r6**	

r6=r4*2 ?p1	r5=r2+r4 ?p1	UdPred r4 [T]	UdPred r7 [R]
r8=r6+r7 ?p3	ST M r6		
r9=r11+r8?p3			
ST (r9) r8			

Figure 5.17: Static Recovery for Example of Figure 5.12

This dissertation uses the latter approach for static recovery of a single recovery block because of its better performance in terms of code generation. Figure 5.17 shows the VLIW schedule for the example of Figure 5.16 using a single recovery block. Within the schedule, the speculated operations are lightly shaded while the non-speculative operations, which depend on predicted values, are highlighted using a darker shading. The scheduler chooses all operations dependent on the loads to be speculated except for the stores (operations 10 and 11) which are non-speculative. The predictions are verified using compare and branch operations that transfer control to the recovery code upon a misprediction. The recovery block consists of all speculated and non-speculated operations. The speculated operations are predicated using the predicates generated during prediction verification. The performance of value prediction using static recovery is analyzed in Section 6 along with a comparison with dynamic recovery.

5.4.2 Dynamic Recovery

Although static recovery allows the restorer to be completely implemented within the compiler, there are some restrictions as a result of a complete software implementation. Upon a misprediction, the control engine branches to the appropriate recovery block and after the recovery code is executed, control is transferred back to the original code. Due to the lack of any parallel execution between these two code forms, the original VLIW code waits for any prior recovery code to finish execution whenever a misprediction occurs. Another effect of software recovery is the frequent transfer of control due to additional branches in the code. The branches incorporated to perform recovery on a misprediction cannot be eliminated since the recovery code is executed only after verification of the prediction. Whenever control is transferred to recovery blocks, the instruction cache would be affected by these blocks. In order to accommodate the recovery blocks, the cache may evict other useful blocks depending on cache replacement policies. Finally, in case of multiple predictions in a region, static recovery either involves redundant computations, in case of a separate recovery per prediction, or additional predicate computations when creating a single recovery block for all predictions in a region.

Hence it is expected that value prediction would suffer some performance loss in VLIW machines when it is performed aggressively and consequently when recovery code is executed frequently. As we show later through experiments, the loss would be even more significant for wider machines. This dissertation also develops an alternate implementation of the restorer by designing specialized hardware and generating recovery code at run-time as follows.

- Recovery code is generated dynamically during execution of the VLIW code with value-speculated operations. Whenever value-speculation is performed at run-time, the recovery code is produced and stored in a separate buffer called the **Compensation Code Buffer (CCB)**. Hence the recovery code need not be scheduled statically with the rest of the VLIW code. This design avoids code growth and does not impact on the instruction schedule. Also, since the recovery code is not executed with the statically scheduled VLIW code, the instruction cache is not corrupted by recovery blocks.

- A dedicated hardware mechanism is provided in the architecture for the execution of recovery code. A separate engine, the **Compensation Code Engine**, executes in parallel with the original **VLIW Engine**. The two engines are synchronized so that the VLIW Engine stalls upon a misprediction only if it requires a value that is yet to be computed by the Compensation Code Engine. Otherwise the VLIW Engine is allowed to continue execution while the Compensation Code Engine recovers from any misprediction by re-executing any mispredicted operations in parallel. In this way, the impact of the recovery code on the execution of scheduled VLIW code is minimized by executing the two engines in parallel. Also, additional branches are avoided by having a separate hardware mechanism that does not function based on control transfer from the VLIW engine. Instead, the execution is synchronized with the static VLIW code and, in this way, the instruction cache is not affected.

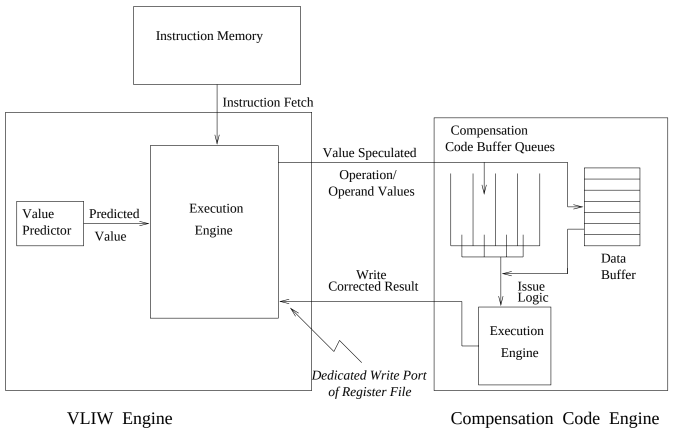


Figure 5.18: High-level view of Architecture with Dynamic Restorer

Figure 5.18 shows a high-level view of the architecture consisting of two separate engines, executing in parallel. The VLIW Engine shown in the figure fetches instructions from memory, decodes them and then executes them on its execution engine. An instruction predicting a value accesses the predictor for the predicted value and enables other operations to use the predicted value for speculation. In Figure 5.18, when a speculated operation is encountered by the VLIW Engine, the operation is executed on the VLIW Engine. At the same time, the decoded operation is sent to the Compensation Code Engine for buffering and re-execution later if the speculatively computed value is found to be incorrect. A prediction is eventually verified by the VLIW Engine when the original operation, whose destination operand was predicted, is executed and the result is compared with the predicted value. If a misprediction is detected, the correct values of the speculated operations are computed by the Compensation Code Engine. The VLIW Engine may stall after detecting a misprediction if the correct values have not yet been produced by the Compensation Code Engine. This situation occurs when an instruction that is ready for execution consists of a non-speculative operation that depends on a value which was previously predicted and the prediction has not been verified yet. When all the operands of such an operation are verified to be correct, the instruction is issued for execution.

The Compensation Code Engine buffers the speculated operations in the CCB for execution in case of a misprediction. These operations have already been decoded by the VLIW Engine. These decoded operations are executed on a simple pipeline engine in the order they were fetched. Execution of an operation on the Compensation Code Engine depends on whether or not the predictions made with the operands are verified as being correct by the VLIW Engine. Hence, an operation is not executed on the Compensation Code Engine pipeline until the outcomes of all predictions related to its operands are verified. Information of a prediction outcome is sent by the VLIW Engine when it executes the original predicted operation and compares the result with the predicted value. If prediction of any operand was incorrect, the associated operation is executed on the Compensation Code Engine pipeline and results are sent to the VLIW Engine. If all operands were predicted correctly, the operation is flushed out of the Compensation Code Engine pipeline since the VLIW Engine had already executed this operation with correct operands. The Compensation Code Buffer consists of a number of queues. Each value accessed from the value predictor is associated with a queue. Operations speculated using such a predicted value are assigned to its corresponding queue. If an operation is speculated by multiple predictions, it is assigned to the queue corresponding to the prediction which is verified latest among the predictions. The reason is that misprediction of such an operation cannot be initiated until all the associated predictions are verified.

The synchronization of the two engines is performed using a mechanism that specifies which operand values are being predicted in every cycle. This mechanism is modeled as a register storing a bit for each value that is speculatively used. This register is called the **Synchronization register** since it needs to synchronize the definition and usage of predicted values. In order to model predictability, every speculatively computed value is assigned a bit within the Synchronization register. A speculatively computed value can be a predicted value coming from the predictor or any value computed by speculated operations. Each speculatively computed operation sets a bit within the Synchronization register to indicate the associated value is not verified yet. The bit is referred to as the *synchronization bit* for the value. The bit value is eventually cleared when the prediction is verified and the correct value is computed, if necessary. In case of a correct prediction, clearing of the bit is done by the VLIW engine after verifying the prediction. For an incorrectly computed value, the Compensation Code Engine clears the bit for the value after computing its correct value. Instructions within the two engines are executed or stalled based on the bits of the

Synchronization register that correspond to the instruction operands. In order to do so, the instructions are appended with bit indices of this register. The indices associated with each operand are pre-determined statically by the compiler. The format of the instructions are elaborated in the next section, which describes the speculation transformer.

5.4.2.1 Speculation Transformer

The speculation transformer exploits value prediction by speculating operations dependent on the predictions in the instruction schedule. This method is similar to the static case with the difference that the recovery code need not be generated at compile-time. The Instruction Set Architecture(ISA) is extended to allow speculation of values. As in the static case, an operation for predicting values is used and other operations are classified as being speculative or non-speculative. Also, there is an operation that checks the prediction and updates the predictor table. Also, operations are annotated with additional bits for synchronization of the recovery code. These operations are described below. As in the static case, each operation type displays operands in <> and any additional annotated bits are in parenthesis.

- Load Predict ($LdPred < r >< TableIndx > [Sync][Queue]$): This operation is similar to the operation used in static recovery for reading the predicted value. The additional information here is the bit index of the Synchronization register ($Sync$). This bit index is set to indicate the corresponding value is predicted and not verified yet. The operation is associated with an identifier $Queue$ which is the queue within the Compensation Code engine assigned to this prediction. This queue is initialized to hold any operations speculated using the predicted value for re-execution in case of a misprediction. The other difference from the static case is that this operation is always used, even while accessing the last value predictor partition. The execution is required for all cases because of the queue initialization and to set the corresponding Synchronization bit. When performing last value prediction, similar to the static case, the speculated operations dependent on this prediction can be initiated in the first cycle using the table index as a source operand. Also, this operation updates the confidence values of the predictor when accessing the hybrid partition.

- Speculative operations ($< Opcode >< dest >< src_1 >< src_2 > [Sync][Queue][Spec]$): The format of a *speculative* operation includes a field corresponding to bit indices of the Synchronization register. The first bit $Sync$ is the synchronization bit of the computed value, which is set by this operation. The other bit $Queue$ corresponds to the queue within the Compensation Code Buffer where the operation is placed. The *Spec* bit specifies the operation to be speculative.

- Non-speculative operations ($< Opcode >< dest >< src_1 >< src_2 > [Sync_1][Sync_2][Spec]$): These operations require correct operand values for execution. Each non-speculative operation is associated with a pair of bit indices of the Synchronization register. These bit indices correspond to the source operand values of the operation. The operation is not executed until the bits corresponding to these indices of the Synchronization register are reset indicating the source operands to be correct. Any predicted value is eventually verified (and corrected if necessary), and the associated Synchronization register bit is cleared. Before issuing a VLIW instruction, all the bits of the Synchronization register for non-speculative operations within the instruction are checked whether any of these bits are set and in that case, the entire instruction is stalled. Also, the *Spec* bit in the instruction format is set to zero indicating the operation to be non-speculative.

- Check Predict ($Ch < dest >< mem >< pred > [SyncIndices][TableIndx]$): This operation performs the verification of the prediction and the predictor update. The operation executes the original load operation from memory location $< mem >$, compares the result with the predicted value $< pred >$ and sends a signal to the compensation code engine. Upon a misprediction, the predictor table index $TableIndx$ is accessed to update the predictor. This operation substitutes the original load operation and incorporates the comparison. The motivation of using this operation is to avoid branches and instead use hardware to synchronize recovery. The operation has latency one cycle greater than the latency of the load because of the additional check. Upon execution, this operation clears bits within the Synchronization register ($SyncIndices$) that are related to the predicted value. These bits include one for the *LdPred* predicted value, which is always cleared after checking prediction regardless of the comparison result, since the correct value is computed along with the comparison. Besides this bit, this operation may clear bits that correspond to the operations speculated using the *LdPred* predicted value. These bits are cleared if the comparison is found to be successful. In case the comparison is not successful, the Compensation Code Engine would compute the correct values for these speculated operations and clear their bits.

Figure 5.19 shows the usage of these operations. This figure corresponds to the example of Figure 5.12. For each operation, the operands are shown along with annotated

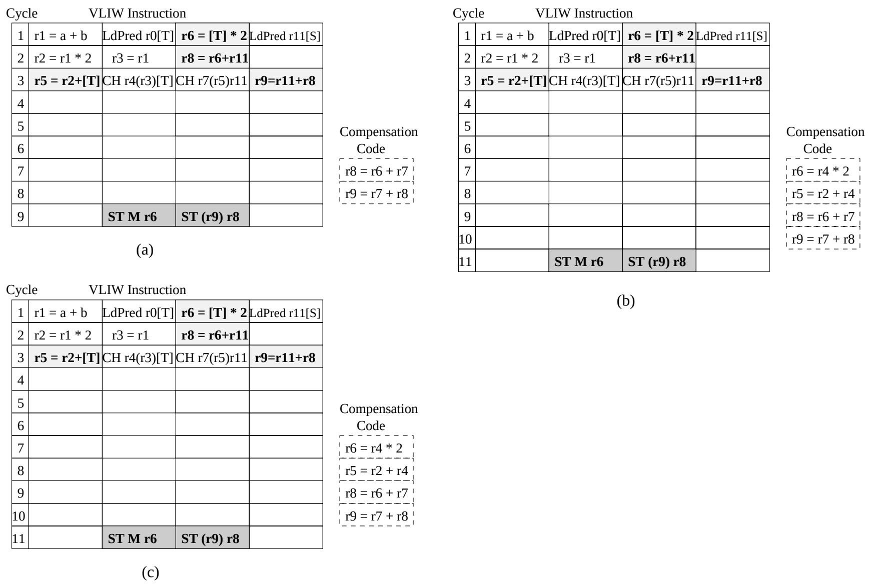
Cycle	VLIW Instruction			
1	r1 = a + b	LdPred r0 [T] [1] [Q1]	**r6 = [T] * 2 [3] [Q1]**	LdPred r11 [R] [2] [Q2]
2	r2 = r1 * 2	r3 = r1	**r8 = r6+r11 [4] [Q2]**	
3	**r5 = r2+[T] [5] [Q1]**	CH r4(r3)[T] [3,4,5,6] [T]	CH r7(r5)r11 [4,6] [R]	**r9=r11+r8 [6] [Q2]**
4				
5				
6				
7		**ST M r6 [3]**	**ST (r9) r8 [4, 6]**	

Figure 5.19: Example of Figure 5.12 incorporating prediction with Dynamic Recovery

bits. In this figure, the two values are predicted using *LdPred* operations. Each of these operations initialize a compensation code buffer queue (shown as $[Qn]$) and set a bit within the Synchronization register, shown as $[n]$. The speculated operations (in lighter shading) access a particular queue to place a copy of the operations. They also store the synchronization bit of the computed value ($[n]$). In cases when the speculated value depends on both the predictions (operations 8 and 9), the queue that stores the operations is the one corresponding to the latest verification. In this example, both the check predictions are scheduled in the same cycle. In such cases of multiple checks qualifying as the latest, the check scheduled right-most within the instruction cycle is assumed to be the latest verification. Hence, in the example, the check prediction testing r7 is assumed to be the latest verification for operations 8 and 9. The non-speculative operations (occurring in darker shading) incorporate synchronization bits corresponding to their source operands. The schedule in the example has an improvement over the original schedule with no value speculation by 2 cycles.

Figure 5.20 shows the dynamic schedules of the example of Figure 5.12 that would result if loads were mispredicted. In case both the predictions are correct, the schedule at run-time is the same as Figure 5.19. Besides the VLIW schedule, the figure also displays the code executed on the Compensation Code Engine pipeline. The annotation bits have been omitted from the schedules for readability. Let us consider each schedule one by one.

Figure 5.20(a) shows the schedule for the case when the value of $r7$ is predicted incorrectly. The misprediction is detected in cycle 6 when the *check predict* operation for loading register $r7$ completes execution. The prediction for register $r7$ had resulted in speculation of other operations. These need to be recomputed by the Compensation Code Engine. Execution within the Compensation Code Engine does not begin until cycle 7 since the *check predict* operations complete execution in cycle 6. Execution on the VLIW instruction is stalled until the Compensation Code Engine computes the values of register $r8$ and $r9$ (required by the non-speculative operation in cycle 9). Note that although


(a)

Cycle	VLIW Instruction			
1	r1 = a + b	LdPred r0[T]	**r6 = [T] * 2**	LdPred r11[S]
2	r2 = r1 * 2	r3 = r1	**r8 = r6+r11**	
3	**r5 = r2+[T]**	CH r4(r3)[T]	CH r7(r5)r11	**r9=r11+r8**
4				
5				
6				
7				
8				
9		**ST M r6**	**ST (r9) r8**	

Compensation Code
r8 = r6 + r7
r9 = r7 + r8

(b)

Cycle	VLIW Instruction			
1	r1 = a + b	LdPred r0[T]	**r6 = [T] * 2**	LdPred r11[S]
2	r2 = r1 * 2	r3 = r1	**r8 = r6+r11**	
3	**r5 = r2+[T]**	CH r4(r3)[T]	CH r7(r5)r11	**r9=r11+r8**
4				
5				
6				
7				
8				
9				
10				
11		**ST M r6**	**ST (r9) r8**	

Compensation Code
r6 = r4 * 2
r5 = r2 + r4
r8 = r6 + r7
r9 = r7 + r8

(c)

Cycle	VLIW Instruction			
1	r1 = a + b	LdPred r0[T]	**r6 = [T] * 2**	LdPred r11[S]
2	r2 = r1 * 2	r3 = r1	**r8 = r6+r11**	
3	**r5 = r2+[T]**	CH r4(r3)[T]	CH r7(r5)r11	**r9=r11+r8**
4				
5				
6				
7				
8				
9				
10				
11		**ST M r6**	**ST (r9) r8**	

Compensation Code
r6 = r4 * 2
r5 = r2 + r4
r8 = r6 + r7
r9 = r7 + r8

Figure 5.20: Run-time schedules of Example of Figure 5.12 (a) Schedule for operation 7 mispredicted (b) Schedule for operation 4 mispredicted (c) Schedule for both predictions incorrect

the schedule of the VLIW code is statically computed, the stalling and issuing of VLIW instructions depends on run-time prediction and verification of values.

In Figure 5.20(b) the value of register $r4$ is mispredicted. The VLIW Engine is informed of this misprediction at the end of cycle 6, when the *check predict* operation form of loading register r4 completes execution. The recovery code in this case is larger, since the number of value-speculated operations that are data dependent on $r4$ is greater than the ones dependent on $r7$. Figure 5.20(c) shows the code in case both the values were mispredicted. Here, the code executed on both the engines is same as the code executed for the previous case. The reason is since the recovery code is the same whether the load operation 4 was mispredicted or both the load operations were mispredicted. From the example, it can be seen that whenever prediction is performed, the total cycle count is better than the case when code is re-executed using static recovery (Figure 5.12).

5.4.2.2 Dynamic Restorer

This section describes the restorer component in detail. This framework component is implemented in hardware and includes the Compensation Code Engine along with slight modifications in the original VLIW engine. Details of these engines are given below.

Primary VLIW engine: The VLIW module of the architecture is similar to a conventional VLIW machine and is shown in Figure 5.21. The additional hardware includes the Synchronization register within the register file, as shown separately in the figure. Upon

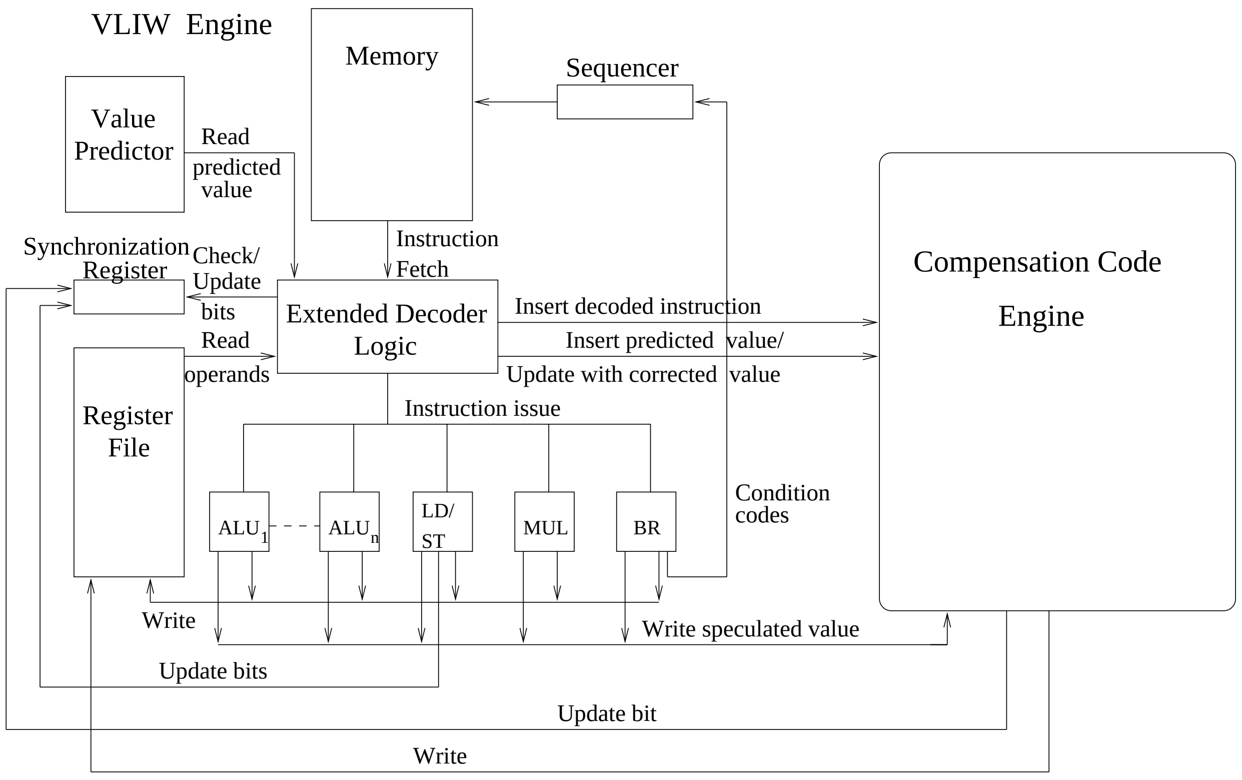


Figure 5.21: VLIW Engine Architecture

fetching a VLIW instruction from memory, the extended decoder logic decodes and executes the individual operations within the VLIW instruction. In case any operation is a *LdPred* operation, the predicted value is fetched from the value predictor. The *LdPred* operation is not issued to any functional unit since it uses a predicted value. The predicted value is sent to the Compensation Code Engine to be stored for any operation to use later. This *LdPred*'s predicted value is later verified by the corresponding *check predict* operation and updated if the prediction was incorrect. Speculated operations using the predicted value are sent to the CC engine. In case the decoder encounters an *LdPred* or speculated operation, it sets the corresponding synchronization bit. Upon computing a predicted or speculatively computed value, the VLIW Engine also sends the result to the Compensation Code Engine. This allows the Compensation Code Engine to read these values locally without accessing the register file of the VLIW Engine. If the value computed by a speculated operation is incorrect, the Compensation Code Engine computes the correct value and writes to the register file through a dedicated write port.

If there are any non-speculative operations in the current VLIW instruction, these operations can only be issued if their operand values are verified and correct. These values are found to be verified when the synchronization bits for the source operands are cleared.

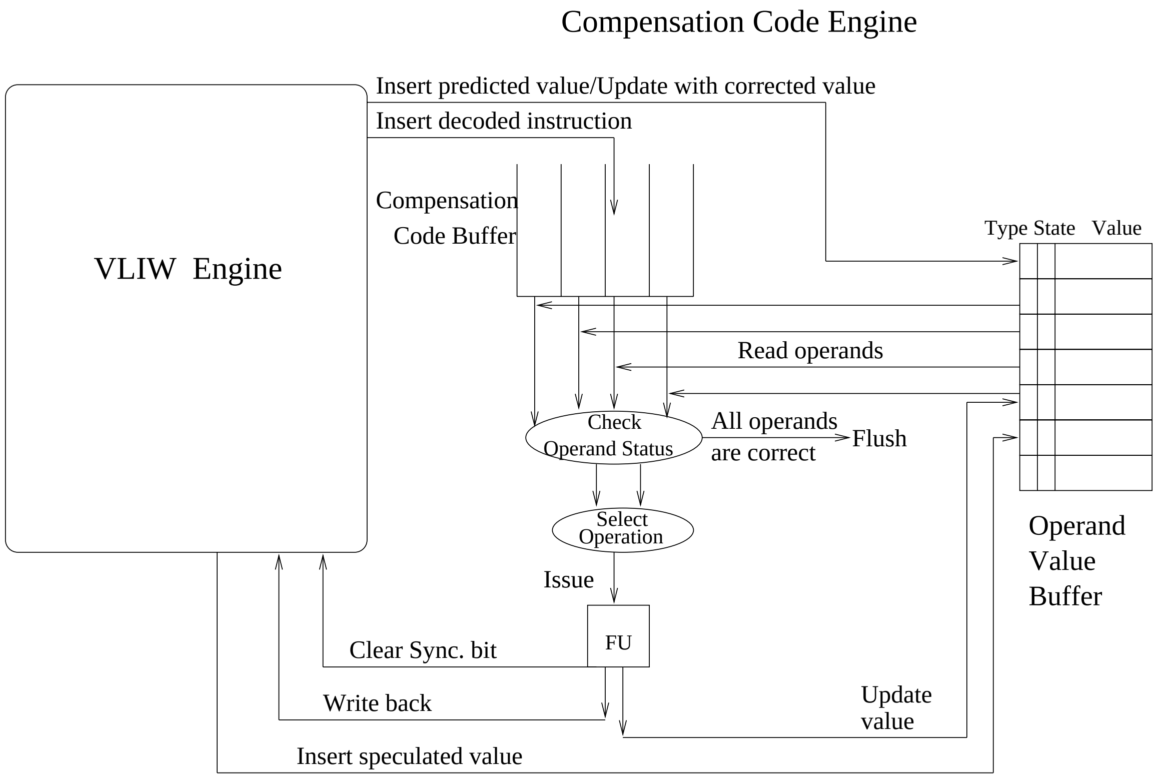


Figure 5.22: Compensation Code Engine Architecture

These bits are cleared by either a *check predict* operation or by the Compensation Code Engine. If the decoder encounters an operation that has a *check predict* form, the operation is issued to a memory unit for loading the correct value. The memory unit is enhanced to provide the capability for performing a comparison. Upon completion of execution, the computed value is compared with the *LdPred* predicted value stored in the register file. If the comparison is successful, the synchronization bits for the values computed by speculated operations dependent on the *LdPred* predicted value are cleared. If the comparison is unsuccessful, the register file is updated with the correct value. The synchronization bit corresponding to the *LdPred* predicted value is cleared in either case. Also the comparison result and correct value are sent to the Compensation Code Engine to update the predicted value stored there.

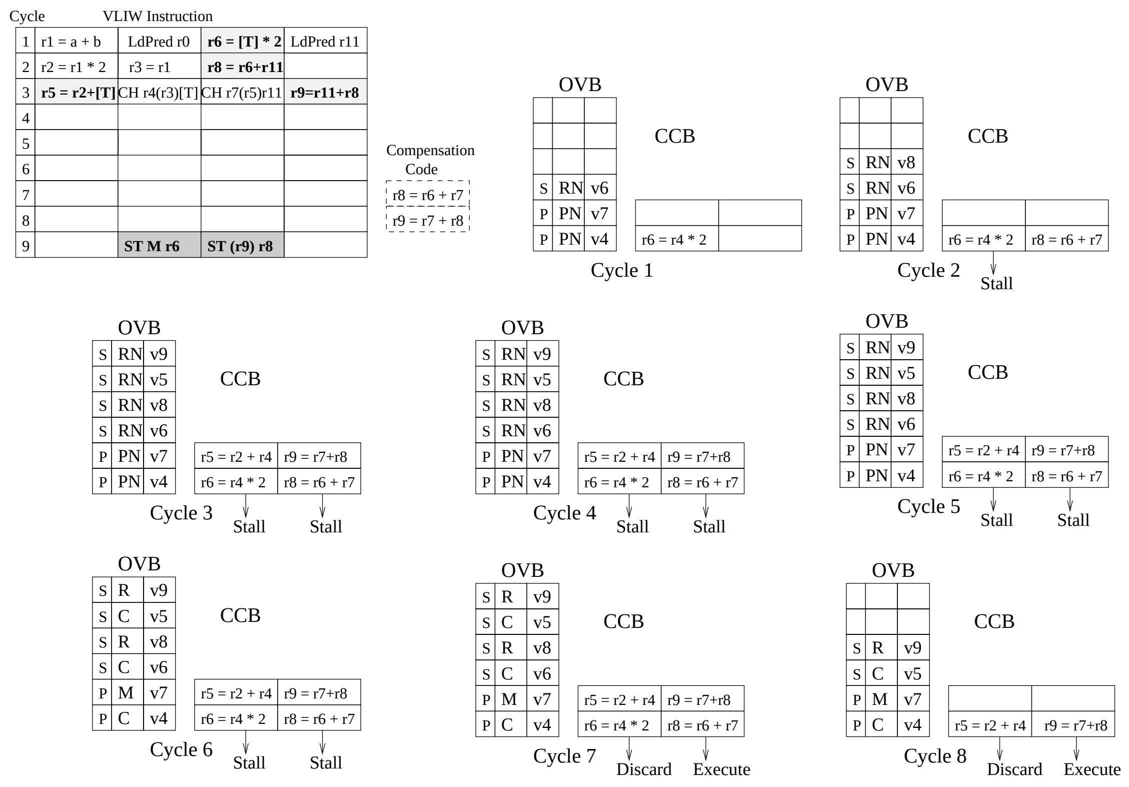
Compensation Code Engine: The Compensation Code module incorporates a simple engine that performs in-order execution of speculated operations. The order of execution of the operations is the same as the order in which they are received from the VLIW Engine. The components are shown in Figure 5.22. The speculated operations are inserted in this buffer as they are executed on the VLIW Engine. In the design of the Compensation Code Engine, four queues have been provided within the buffer allowing at

Operand Type	*Initial State*	*Final State*	*Condition*
Predicted using *LdPred*	Prediction not verified	Mispredicted	Operand value mispredicted
		Correct	Operand value correctly predicted
Predicted by being speculatively computed	Recomputation not verified	Recompute	Associated prediction incorrect
		Correct	Associated prediction correct
Correct	Correct	Correct	Always

Table 5.4: State Table of Operand within Compensation Code Engine

most four active predictions at the same time. More than 90% of the regions had the number of useful predictions less than or equal to four and hence the number of queues was restricted to this number. In case of regions with useful predictions exceeding four, only the first four predictions within the region were made. Upon a procedure call, the contents of the buffer are saved on the stack. During every cycle, the head of each non-empty queue is analyzed for execution. Operations that are inserted in the CCB may not always get executed by the Compensation Code Engine. They are only executed in case they were incorrectly speculated and the synchronization bit is also cleared in this case. If the operand values of a speculated operation were correctly used within the VLIW engine, the Compensation Code Engine does not need to re-execute the operation. Instead, the operation is flushed as shown in the figure. Hence, the values of the operands are always sent along with the operation but these values may or may not be used by the Compensation Code Engine. In the design of the Compensation Code Engine, a single functional unit capable of executing ALU operations is provided allowing one operation to be recovered per cycle. In case more than one operation is eligible for re-execution, the operation corresponding to the earliest prediction is chosen. In order to do so, each queue is stamped with the time it is initialized.

Each operand is assigned a state depending on how it is computed and whether its value is correct or not. An operation is issued for execution based on the states of its operands. Table 5.4 shows an operand can be either predicted using *LdPred*, predicted by being speculatively computed, or correct. If the operand is predicted using *LdPred*, it is assigned an initial state of not being verified. Once the associated *check predict* operation is performed, the operand's state is marked as either correct or needing recomputation, depending on whether the prediction is correct or not. If an operand is predicted by being speculatively computed, the initial state denotes that it is not known yet if the operand's value needs to be recomputed within the Compensation Code Engine. When the associated *LdPred* prediction is verified, the state is updated to either correct or marked for recompu-



Cycle	VLIW Instruction			
1	r1 = a + b	LdPred r0	**r6 = [T] * 2**	LdPred r11
2	r2 = r1 * 2	r3 = r1	**r8 = r6+r11**	
3	**r5 = r2+[T]**	CH r4(r3)[T]	CH r7(r5)r11	**r9=r11+r8**
4				
5				
6				
7				
8				
9		**ST M r6**	**ST (r9) r8**	

Figure 5.23: VLIW Schedule and Execution Trace

tation. The last case occurs when the operand does not involve any prediction, directly or indirectly, through any dependence and its state is marked as being correct.

The issue logic of the Compensation Code Engine avoids execution of any operations other than those required to recover from mispredictions. To execute only the required operations, each operation is assigned a state of either being stalled, issued for execution or flushed. These states are derived directly from the states of the operands. If any of the operands is in a state of not being verified, the operation waits for the correct values before being issued for execution. This implies the Compensation Code Engine stalls if an operand's value is not verified yet. An operation is issued for execution if none of its operands is in the state of not being verified and if one or more of its operands is in a state requiring recomputing. The latter condition happens if any of the operand's value was mispredicted leading to the operand's state being marked for recomputing. In such a case, the operation would need to be executed locally with the correct operands. If all operands are in the correct state, the operation is flushed since the correct values would already have been computed by the VLIW Engine.

In order to store the values needed by the Compensation Code Engine, a buffer is provided in the architecture called the **Operand Value Buffer (OVB)**. It stores the operand values as well as bits for the type and state associated with each operand value. An entry is inserted into the buffer when the VLIW Engine sends the decoded operation along with the operand values. An entry is updated on different conditions for the different operand types. If the operand is predicted, its entry gets updated by the VLIW Engine when the prediction is verified. For an *LdPred* predicted value, the update is for both the value and state. For a predicted operand that was speculatively computed, the state update is similar but the value is corrected upon a misprediction only after the operation computing the value is executed within the Compensation Code Engine. An operand value that is correct does not need to be updated.

The operations are initially fetched from the CCB and the operand values are read from the OVB. The operands are checked to see whether the operation needs to be executed. The operation behaves according to the states of the operands, as mentioned above. After execution of an operation on the recovery code pipeline, if the computed value is required by a subsequent operation, it is written to the OVB (besides being always written back to the register file of the VLIW Engine).

The entries of the OVB can be traced for the Figure 5.20 example. Figure 5.23 considers the case of Figure 5.20(c), when the value in register r4 is correctly predicted and the value in register $r7$ is mispredicted. The scheduled VLIW code along with the recovery code is shown to the left in the figure. The contents of the OVB and CCB are shown next to each other for each cycle. Each entry of the OVB is labeled as v_n where n is the register number storing that value. Hence $v8$ is the value computed and stored in register $r8$. Note that there may be more than one speculated value for a particular register but this is not the case in the example shown. To make the figure more readable, we do not show the correct entries (having no prediction) in the OVB. The speculated operands have the state S while the predicted operands have the state P. Also the states from Table 5.4 are denoted as follows. State PN denotes the state of the operand's prediction to be not verified while state RN is the state of the operand's recomputation not verified. State C is the state of the operand's value being correct while R is the state requiring recomputation.

In the figure, the contents of the CCB and OVB are shown after execution of each cycle. Before execution, there are no entries within these buffers. When the first cycle completes, the values of register $r4$ and $r7$ are sent to the Compensation Code Engine and stored in the OVB. These values are assigned the status of not being in the verified state (PN). Also, a copy of the speculative operation computing $r6$ is added to the Compensation Code buffer. This operation speculatively computes register $r6$ within the VLIW Engine and stores the value in the Value buffer. A state value of RN relates to the default initial state for the speculated value. Similarly, the speculated values of the registers $r5$, $r8$ and $r9$ are added to the OVB in the next two cycles and the corresponding speculated operations are placed in the CCB. Notice that two separate queues corresponding to the two predictions are used for storing the operations. The first queue stores the operations dependent on $r4$. Operations computing $r8$ and $r9$ depend on both predictions. They are placed in the queue for $r7$'s prediction because an operation is stored within the queue of the prediction which is verified last[2].

In cycle 6, the predicted values of register $r4$ and $r7$ are verified and their corresponding states in the OVB are updated. Since $r4$ is correctly predicted, the state is updated to that of being correct. The speculated values $v5$ and $v6$ are dependent on $r4$ and are deemed to be correct. On the other hand, the value of $r7$ is found mispredicted and its state is updated as mispredicted. The dependent values $v8$ and $v9$ are set as requiring recomputation and subsequent cycles re-execute each one of the two operations. These values are sent to the VLIW Engine to allow any stalled non-speculative operations to execute. In this example, the store operations are non-speculative and eventually execute in cycle 9. On the other hand, the correctly computed operations ($v6$ and $v5$) can be discarded in parallel with the recomputation in cycles 7 and 8 because they are present in a different queue.

5.5 Experimental Results

To evaluate the efficacy of the value prediction framework, experiments were performed to study the performance of a conventional VLIW machine. The underlying machine model used was a 8-wide VLIW consisting of four ALUs, two memory units, one floating-point unit and one branch unit. The machine used 128 general purpose registers along with 64 floating-point and 128 predicate registers. The value predictor was implemented as explained in Section 5.4 and consisted of 256 entries partitioned equally into last value and hybrid predictor components. The second-level table consisted of 16K entries. The threshold of load prediction (from value profile) was kept at a percentage of 65%. Each predicted load was allowed to speculate up to four dependent operations. The performance of the value prediction framework was evaluated for both the static and dynamic recovery cases.

[2]although the rightmost check within a VLIW instruction is assumed to be the latest among all the checks within that instruction, it is possible to use any other policy, such as left-most check, without any loss in efficacy

Benchmark	*Best case*	*Worst case*
SPECint95 programs		
compress	0.91	1.07
li	0.87	0.99
m88ksim	0.89	0.99
ijpeg	0.92	1.13
vortex	0.80	1.03
Mediabench programs		
epic	0.91	1.15
rasta	0.92	1.05
mpeg2dec	0.95	1.05

Table 5.5: Schedule lengths of blocks with speculated operations as a fraction of schedule length with no prediction

Benchmark	*Ratio for issue width=8*	*Ratio for issue width=16*
SPECint95 programs		
compress	0.91	0.83
li	0.87	0.73
m88ksim	0.87	0.78
ijpeg	0.92	0.84
vortex	0.80	0.71
Mediabench programs		
epic	0.91	0.81
rasta	0.92	0.90
m88ksim	0.95	0.83

Table 5.6: Schedule length of best case for different issue widths

5.5.1 Dynamic Recovery

In case of dynamic recovery, the efficacy of the modified architecture with the compensation code engine was initially evaluated. This evaluation was done by observing the schedule lengths of the blocks due to value prediction. Table 5.5 shows the effective schedule lengths of the blocks incorporating prediction (calculated after including cycles for recovery code) as a fraction of the schedule lengths of the original blocks (with no predictions). The best case column indicates the case when all predictions within a region were correctly predicted, and the worst case gives the case when all predictions within a region turned out to be incorrect. As an example, for the *compress* benchmark, the schedule length in regions where predictions were made, reduced by 9% on average when all of the predictions were correct. For all of these regions, the schedule increases by 7% on average

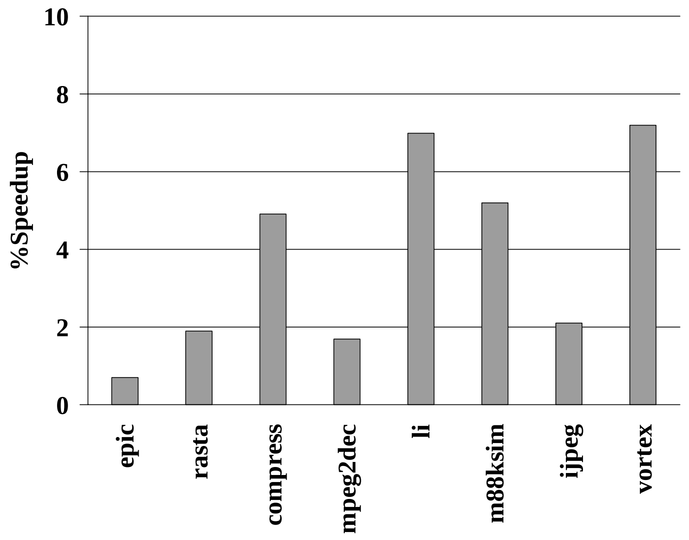


Figure 5.24: Performance for Dynamic Recovery

when all the predictions were incorrect. In the best case with all correct predictions, the schedule length reduces by about 15% on an average. In the case when all of the predictions within a block turn out to be incorrect, the schedule still manages to improve for some of the cases. This improvement is due to the the additional execution power of the compensation code engine allowing recovery code to execute in parallel with the original VLIW code.

The machine width used for the above results was varied to observe performance for different architectures. The effect of recovery code was also analyzed on a 16-wide VLIW machine, and Figure 5.6 shows the best case for each benchmark. From the figure, we observe that for the higher issue width, the improvement in block schedule length is higher. This result occurs because in order to create greater degrees of parallelism, an increased amount of speculation is performed. This observation is similar to previous studies in the superscalar domain [34, 31] that observe the benefits of value prediction to increase as the instruction window grows. In the wider machine, it was also observed that the processor spent less time in executing the correctly predicted code and the likelihood of mispredictions and execution of recovery code increased. Hence, the impact of executing recovery code on overall performance is higher for a wider machine. This observation reinforces our earlier assertion that the effect of recovery code is more significant for wider machines.

Performance speedups were computed by computing the cycle count in the simulator and comparing against the case when no value prediction was applied. In both cases, programs were compiled by turning on all optimizations. Value speculation was performed along with the other optimizations in the compiler back-end, and prior to the scheduling and register allocation phases. Figure 5.24 shows the performance speedups using dynamic recovery. From the figure, it is observed that speedups are mostly in the range 1-7%. The integer benchmarks benefit more than the Mediabench benchmarks. In the latter case, sufficient instruction-level parallelism exists and there are not enough opportunities to extract additional parallelism. As we shall see in the next chapter, these benchmarks are more suitable for overcoming resource dependencies when the machine model cannot handle the parallelism available in the program. Amongst the integer benchmarks, the performance for most of the benchmarks is in the range 5%-7% with the exception of *ijpeg*. The latter benchmark does not perform as well as the others. The reason is that the amount of register spills is higher in case of *ijpeg* than the other benchmarks shown. These spills subsume some of the performance benefits of value prediction. Register spills are a concern in all benchmarks since value speculation increases the amount of parallelism, and thereby increasing register pressure. In case of *ijpeg*, the effect of spills is most noticeable. An integrated approach that performs register-sensitive value speculation would reduce the impact of register spills on the performance. One of the limitations of performance in case of dynamic recovery is that recovery was kept to one operation per cycle. The compensation code engine was designed to be simple with a single functional unit. With more sophisticated designs, the performance of dynamic recovery can be further increased by recovering multiple operations in parallel.

The additional functional unit used in the compensation code engine is utilized for handling speculative instructions. Another way to use extra functional units within the architecture is by re-compiling the entire program and executing it non-speculatively on the wider machine. The latter case would not be beneficial for wide VLIW processors, which are the primary focus of the value prediction optimization. The reason is that a wide machine already has sufficient resources for execution and its performance critical path is dominated by the dataflow barrier within programs [34, 31]. This concept is the motivation behind using extra functional units for dynamic recovery in value prediction. The VLIW machine incorporating dynamic recovery was compared to a machine model using an extra ALU for executing non-speculative code. The performance of the former model was better by 2-3% in terms of speedups suggesting that incorporating the compensation code engine is a more appropriate usage of additional functional units. This observation is expected to be more significant as the machine width increases for next-generation VLIWs.

5.5.2 Static Recovery

The performance of the prediction framework using static recovery is shown in Figure 5.25. For this case as well, value prediction was applied along with the entire suite

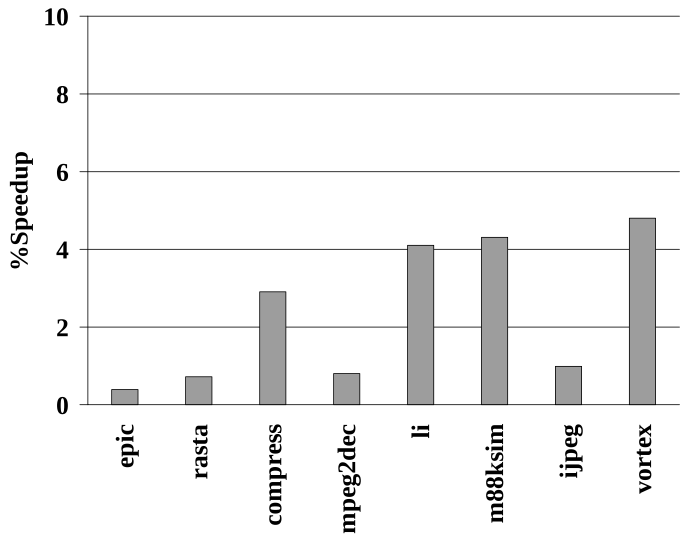


Figure 5.25: Performance for Static Recovery

of back-end optimizations, prior to scheduling and register allocation. From the figure, it is observed that performance is not as good as the dynamic recovery. The speedups are lower in this case averaging 3-4%. One of the reasons why static recovery does not measure up to the dynamic case is because the latter involves better recovery handling. As elaborated in Section 5.52, run-time generation of recovery code has less misprediction penalty and also can handle prediction of multiple loads within a region more efficiently. In most of the benchmarks, the performance degrades by about 2% from the dynamic recovery case. In case of *m88ksim* the speedup using static recovery is almost the same as speedup for the dynamic recovery case. The reason is that *m88ksim* displays high prediction accuracies (as seen from Table 5.3) and hence, the effect of recovery code is not noticeable in either implementation of the restorer. One of the limitations of static recovery is that the underlying compiler infrastructure constrains the performance of value prediction. The compiler uses superblock regions for scheduling and other optimizations to expose greater parallelism than basic blocks. These superblocks may not be the most appropriate regions for static recovery because they lead to fragmentation while generating recovery blocks. Figure 5.14 showed this process of recovery block generation. The overheads involved in executing the additional branches due to this fragmentation subsume part of performance improvements. Hence, unlike dynamic recovery case, the average schedule length fraction for regions with prediction in the worst case (of Table 5.5) was higher than 1.1 for all the benchmarks. One of the advantages of static recovery is that the VLIW processor is not modified unlike the dynamic case. The latter case involves an additional execution engine for recovery. Though the Compensation Code Engine involves an additional ALU for recovery, it is also possible to design it without any additional functional units. In that case, the functional units of the original VLIW engine can be reused for dynamic recovery. The Compensation Code Engine would only consist of a buffer to store the speculated operations at run-time along with control logic to perform the dynamic recovery.

From the above results, it can be concluded that value prediction is able to improve performance of VLIW processors. When mispredictions are recovered dynamically, the improvements are higher than the static case. Mispredictions are observed to be a significant factor in the performance of value prediction. Executing recovery code dynamically restores mispredictions in parallel with the VLIW code and also avoids any overhead of static code size increase. Dynamic recovery involves the architecture to consist of an additional engine executing recovery code besides the conventional VLIW engine. The additional engine is a simplified in-order pipeline engine synchronized in execution with the VLIW engine. The synchronization may incur additional latency which can affect the cycle time. It is possible to prevent cycle time increase by masking the synchronization with other pipeline operations executing in parallel. A low-level simulation of the machine design would determine the exact effects of the additional engine on the pipeline latency. Although performance with static recovery does not measure up to dynamic recovery, there is no hardware overhead in the former case. In case hardware complexity is critical in the VLIW processor, value prediction should be applied with static recovery. The above results also show value prediction to be more significant for wider issue machines.

Chapter 6
Bit-width Prediction Framework

6.1 Motivation

As processors have evolved to support wider instruction words, there has been an increase in the opportunities to optimize data widths. Although newer processors support up to 64 bits of data, the applications running on these processors rarely require the entire data width. There has been some work with compiler optimization and computer architecture to exploit data width by packing several operations together to execute on a single functional unit [8, 42]. Most of this work has focused on superscalar processors utilizing sub-word parallelism to pack similar (homogeneous) operations with narrow operands. This dissertation evaluates packing of multiple narrow operands on VLIW processors. In particular, the prediction framework was used to develop an optimization, called bit-width prediction, that predicts operand width of instructions.

Several commercial processor ISAs provide support for SIMD operations specifically targeted towards multimedia benchmarks. These benchmarks benefit greatly from SIMD synthesis because they typically operate on 16-bit data. Since multimedia applications are common for embedded systems, packing operations to synthesize SIMD-like instructions is very promising in this domain. However, as embedded systems move toward an EPIC-style Very Large Instruction Word (VLIW) computing model, previous work with SIMD synthesis for superscalars is not directly applicable and there is a strong need to address scheduling for narrow data types on VLIWs. The importance of optimization for embedded VLIWs is underscored by the popularity of recent architectures such as Starcore's SC140, Texas Instrument's TMS320C6201, and Transmeta's Crusoe. These processors are resource-constrained due to the stringent cost limitations of embedded systems. By packing narrow width operations to execute on a single functional unit, we can take better advantage of hardware resources. The bit-width prediction technique in this dissertation targets embedded VLIWs with limited hardware resources, although it is also implemented for wider VLIW machine models.

In the context of VLIW processors, exploiting narrow data widths has several unique challenges. These challenges are in addition to the challenges pertaining to the prediction framework which were outlined in Chapter 3. Among these additional challenges is how to detect the width of operands for the compiler to exploit during scheduling. One approach to determine width is by using dataflow analysis to compute bounds on the operand widths. Static bitwidth analysis has recently been proposed in the context of synthesizing reconfigurable architectures to identify the width of operands [72]. Although this information is accurate, it is conservative and cannot consider run-time information such as program input within the analysis. In the bit-width prediction optimization developed in this dissertation, a feedback-directed approach is used to predict the width of operand values and speculatively pack narrow operations on a functional unit. This approach fits well within the prediction framework which is driven by profiling.

A high-level example motivating the use of feedback in bit-width prediction is shown in Figure 6.1. For simplicity, assume all the operations shown to be ALU operations of the same type and the underlying processor to have two 32-bit ALUs. This machine restriction results in the instruction schedule shown on top of Figure 6.1. The scheduled program is initially instrumented to collect profile information about the width of instruction operands. The program is executed with training inputs to gather the profile information. Assume that profiling information indicates operations 3 and 4 to always have 16-bit operands. The program's intermediate representation(IR) is annotated with this information. Next, the compiler uses the annotated IR to generate an instruction schedule that packs operations 3 and 4 on the same ALU. The schedule length is now improved by one cycle due to the packing. Here, bit-width prediction is able to improve the schedule because the underlying machine model cannot handle all the available parallelism within the application. Notice that if additional ALUs were present in the machine, bit-width prediction would not improve the schedule, implying that the technique is more suitable for resource-constrained VLIWs. Since profile information is used for prediction, it is possible that the either of the operands of operations 3 or 4 spill over 16 bits during execution and appropriate recovery mechanisms need to handle such cases.

Another issue in implementing bit-width prediction is whether there are enough opportunities to make static scheduling of narrow width operations worthwhile. In previous work related to operation packing [42, 8], only operations of the same type were combined together for parallel execution on a functional unit. For VLIW architectures, there is a

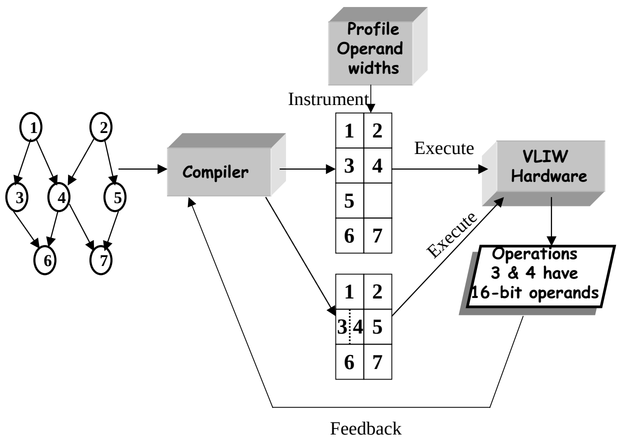


Figure 6.1: High-level view of Feedback used for Bit-width Prediction

unique opportunity to combine narrow width operations of different types without requiring significant hardware modifications. By adding support for heterogeneous operations, bit-width prediction can uncover more opportunities for improving performance. However, the VLIW machine model involves an additional challenge of fetching packed operations in the same instruction word. From Figure 6.1, the schedule in the second cycle after packing requires an additional operation to be fetched. The limited fetch bandwidth of the processor may not allow fetching of additional operations in an instruction word. The fetch bandwidth can be increased to accommodate the additional operations but at the cost of increased processor complexity. Another approach is to create special (SIMD) operations that operate on multiple data streams. However, this approach limits the operations packed together to be of the same type and is unable to exploit the opportunities of packing heterogeneous operations. Hence, the prediction framework needs to address the issue of exploiting operand width effectively without a substantial increase in complexity of the underlying processor.

The following sections describe the prediction framework components built for bit-width prediction.

6.2 Prediction Schemes

The first step in utilizing the prediction framework was to design the prediction schemes for operand widths. These prediction schemes would estimate the predictability of operand widths indicating the potential of implementing bit-width prediction. Although researchers have exploited the width of operands for performance and power, there has been no previous work relating to prediction of operand widths. The advantage of predicting bit-width is to be able to pack operations on the same functional unit based on the predictions. In context of VLIW processors, the instructions are scheduled at compile-time and packing needs to performed statically. Hence, bit-width prediction should be implemented statically for a VLIW processor to allow the compiler to utilize these predictions during scheduling. The effect of static prediction of operand widths is investigated for accuracy. Amongst the functional units, only ALUs were analyzed for packing with multiple operations in this dissertation. It is infeasible to pack multiple load or store operations on a memory unit since the address bus can access only one memory location per cycle. Also, in the benchmarks used for experiments, floating-point operations accounted for a negligible fraction of total operations and hence, were not analyzed for bit-width prediction.

Initially, several benchmark programs were analyzed to observe the size of the operands of instructions. All the benchmarks in the suite used for evaluating the value prediction framework (introduced in Table 5.1) barring *vortex* are used in this chapter for testing bit-width prediction. Bit-width prediction was developed for embedded processors and a large benchmark like the object-oriented database, *vortex*, is not likely to be used in the context of embedded processors. Instead, another Mediabench benchmark, *G721dec* (voice decoder), was added to the benchmark suite. Table 6.1 shows the percentage of integer arithmetic operations that are narrow enough to be packed with other similar operations. The table shows a distribution of 32-bit operations that have all operands within 16 bits and 8 bits during the entire program execution. For each benchmark, the dynamic percentage column shows the fraction of total executed arithmetic operations that have all operands within 8 or 16 bits. The static percentage column shows the percentage of the arithmetic instructions in the static code that always have narrow operands. The benchmarks in the table include SPECint95 programs as well as Mediabench programs. The underlying processor uses a 32-bit wide datapath. From the table, it is observed that a large fraction of operations in the benchmarks have narrow operands. These numbers are not only apparent in the multimedia benchmarks, but also the SPECint95 benchmarks display the potential of finding operations with narrow operands. This observation suggests several opportunities to identify narrow operands for packing on an ALU. When the benchmarks were executed with different inputs, the difference in percentages of narrow operand widths was minor and always less than 5%. This latter observation motivates the use of profile data to predict which operations would typically have narrow width operands.

The granularity of operand size that was analyzed in the above benchmarks is at the byte-level. It is also possible to analyze operand sizes at finer granularity, such as at the bit-level. Finer granularity analysis increases opportunities to pack operations

Benchmark	Percentage of instructions			
	all operands always <= 16bits		all operands always <=8bits	
	Static	Dynamic	Static	Dynamic
SPECint95 programs				
compress	70.1	60.3	35.7	23.5
li	42.7	40.2	34.6	33.6
m88ksim	52.5	64.7	37.8	47
ijpeg	69.1	59.6	40.5	9.1
Mediabench programs				
epic	64.9	68.4	46.5	61.6
rasta	86.8	95.4	63.6	44.3
mpeg2dec	59.5	55.5	32.1	29.0
G721dec	68.5	72.1	30.1	32.9

Table 6.1: Percentage Instructions with Narrow Operands

together. For example, operations with 4-bit operands can be packed to execute upto eight such operations on a 32-bit ALU. However, the information required to encode bit-level predictions within instructions would be substantial because of the large number of possible predicted sizes. Also, the hardware complexity for accessing operand values of sizes smaller than a byte would be quite high. Hence, only byte-level widths were considered for packing in this dissertation. Within byte-level analysis, only 8-bit and 16-bit sizes were considered. Operands that were wider than 16 bits but less than 24 bits do not have enough opportunities for sharing an ALU, except with 8-bit operands (since the underlying functional units are 32-bits wide). Also, upon analysis of programs, it was observed that after honoring all the program dependencies, there were very few cases when operations with operands between 16 and 24 bits could be packed together with other 8-bit operations. Hence, the bit-width prediction technique was limited to packing operands within 8 or 16 bits of an ALU. As we shall see in later sections, this decision also kept the complexity of the hardware and extensions to the instruction-set architecture within reasonable limits.

Hence, predicting operand widths statically displays the potential for utilizing the framework to implement bit-width prediction. The implementation of the framework components was performed on the Trimaran compiler infrastructure, details of which are elaborated in the previous chapter. The following sections discuss the rest of the framework in context of bit-width prediction.

6.3 Usefulness Analyzer

A pre-requisite for applying bit-width prediction is that there should be sufficient operations that can be packed together to improve the instruction schedule. Operations that are expected to have at least one operand large enough to occupy most of the available bits need not be profiled. Also, the regions in which program dependencies sequentialize the execution of operations should be avoided for profiling. Such regions, having sparse parallelism, would typically under-utilize the resources, and bit-width prediction would be more suitable for regions with excessive parallelism. This latter observation is in contrast to the case of value prediction. In case of value prediction, the predictions were used to extract additional instruction-level parallelism within the regions in the program graph that were devoid of parallelism. On the other hand, bit-width prediction allows more operations to execute concurrently, assuming the program graph already displays sufficient parallelism.

In this dissertation, usefulness was evaluated both at the instruction-level and at a larger regional level. In both cases, only frequently executed blocks were evaluated for usefulness.

- *Instruction-level analysis*: As mentioned before, operations are analyzed for packing within 8 or 16 bits of an ALU. In the experiments that were performed, it was observed that the virtual addresses of instruction and data accesses in a program were always at least 16 bits. The reason is because the first 128K locations of memory were not used by the simulator assuming the kernel resided in those memory locations. All memory references in programs were made to addresses in locations higher than 128K and required more than 16 bits for representation. This information was used by the bit-width prediction framework to avoid profiling operations with address operands. Such operations cannot be packed with any other operations on the same functional unit because the memory address operand would require the entire 32-bit ALU (packing was performed at the 8-bit and 16-bit granularity). Several of the operations with memory addresses as operands were add operations calculating the stack pointer during register spills. It is expected that instruction-level usefulness can be applied for most VLIW processors since the lower segments of memory are always reserved by the operating system.

- *Region-level analysis*: Usefulness is also analyzed at the region level by identifying the program regions with insufficient parallelism. Such regions would be devoid of resource dependencies and their performance would not improve by packing operations. These regions can be identified as the ones where the instruction schedule would not improve if additional resources were made available. Packing of operations using bit-width prediction emulates a wider machine by overcoming some resource dependencies. Hence, if a region displays the same schedule length as the length on a processor with additional functional units, the instructions within the region would not be constrained by resource limitations. These regions are identified in the usefulness analyzer by creating a *shadow schedule* during instruction scheduling. This schedule is created for a machine model having one additional ALU than the standard machine model used for evaluating bit-width prediction. Since ALU operations were analyzed for prediction in the experiments, the extra functional unit added was an ALU. The shadow schedule was created in parallel with the instruction schedule created for the original machine, and the two schedules were compared. If a region displayed the same schedule length in both cases, it was not selected for profiling. Since scheduling was performed at the granularity of extended blocks (superblocks), this technique was able to eliminate profiling of entire blocks.

The next section evaluates the effectiveness of the usefulness analyzer by comparing the profile overhead with the case when no usefulness analysis was performed.

6.4 Selective Profiler

Similar to the case of value prediction, feedback information was generated as late of possible within the Trimaran infrastructure to avoid any transformations from corrupting the profile data. Hence, the performance monitoring framework of the simulator was modified to profile operand widths of ALU operations (bit-width profiling). The profiling includes generation of run-time traces and analysis of this information to determine the average size of instruction operands. For each integer arithmetic operation, the width of operands is generated in the trace and these operand widths were analyzed whether they were less than 8-bits or 16-bits using a prediction threshold. Given b bits, a *prediction threshold(b)* is used for analysis, and is defined to be the percentage of times an operand is found to be less than b bits wide. Since operands are either predicted to be 8-bit wide or 16-bit wide, two different prediction thresholds are used for prediction. An operand is predicted as 8-bits wide if it met the prediction threshold for 8-bit prediction from the profile data. Similarly, an operand is predicted as 16-bits wide if it met the prediction threshold for 16-bit prediction from the profile data. The thresholds can be varied to control the degree of operation packing. For a lower threshold, a larger percentage of operations can be packed together. The variation of threshold will be analyzed in context of mispredictions in

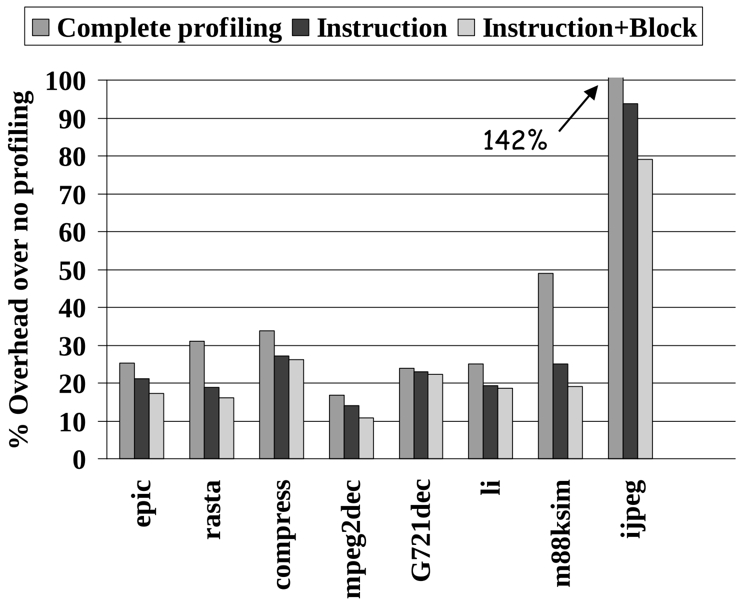


Figure 6.2: Overhead in Profile Time due to Selective Profiling

the section describing the restorer. For the purpose of selective profiling, both thresholds are set to 100% in the analysis of the profiles. Hence, the trace information is analyzed to detect the percentage of instructions whose operands were always within 8-bits or 16-bits.

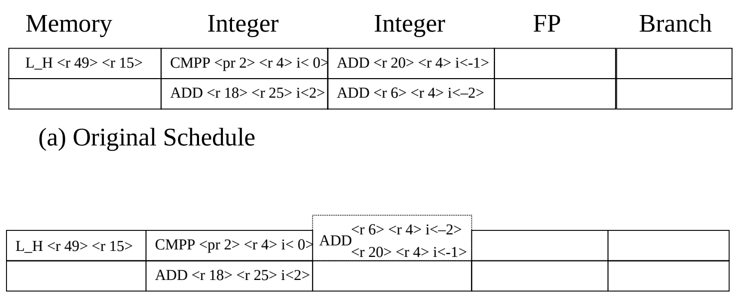
Since profiling is performed within the performance monitoring framework of the simulator, it creates execution overhead. The overhead of profiling was analyzed and is shown in Figure 6.2. In this figure, each benchmark is associated with three bars. The first bar displays the execution overhead if all ALU operations were to be profiled as compared to no profiling. The second bar indicates the overhead when usefulness at the instruction-level is applied. The third bar shows the overhead when usefulness at the region-level as well as the instruction- level is applied. The figure shows that instruction-level usefulness reduces profile overhead by about 10% on average. Region-level usefulness further reduces the profiling overhead by 5% on average. The exception is *ijpeg* where the profile overhead as well as the reduction due to usefulness analysis are higher than other benchmarks. The reason is that spills are a dominant factor in *ijpeg* resulting in a large number of addition operations that compute the stack address for spilling. These operations are eliminated in the usefulness analysis. For all the benchmarks, the usefulness analyzer contributes to reduce the profile overhead to reasonable limits.

6.5 Speculation Transformer

The speculation transformer performs packing of multiple operations on the same ALU. The profile information is used to identify which operations to pack together. An operation is packed within 8-bits of an ALU if all of its operands are predictable to be within 8-bits and similarly the case for 16-bit packing. Each operation is associated with two additional bits *predicted_8* and *predicted_16*. The former bit is set by the compiler if the operation is predicted to have 8-bit operands and the latter is set if the prediction is for 16-bits. These bits are set based on the profile data, and the scheduler uses them to pack operations together. Since only ALUs were packed using bit-width prediction, the ALU operations were annotated with these additional bits.

One of the issues within the speculation transformer is how to pack the operations together in an instruction word. Conventional techniques that exploit operand widths for optimization synthesize SIMD-like instructions. An example from one of the analyzed benchmarks demonstrates such a case. Consider a five-wide VLIW processor with two integer functional units, and assume that the fetch unit is tightly coupled with the functional units. Hence, there is a one-to-one correspondence between the functional units and the fetched operations. Several VLIW processors use this tight coupling to simplify issue logic. In Figure 6.3(a), we show part of a schedule for a region from the *_compress* function in the *compress* benchmark. Each operation is assigned the ALU that would execute the operation. Note that although the instructions are independent, they need two cycles to be scheduled since only two integer operations can be accommodated per cycle. However, profile information indicates that each of the integer operations have operand widths that are at most 16 bits. This information can be used to pack the two add operations on a single ALU and fetch them as a single SIMD operation. Performing this SIMD synthesis yields the schedule shown in Figure 6.3(b). In order to pack the two adds as a SIMD operation, the instruction set architecture (ISA) needs to have the capability to specify pairs of registers in an add operation. The reason is that most of embedded VLIW ISAs do not have vector registers.

One observation from Figure 6.3(a) is that the processor has the capability to fetch five operations per cycle, yet two of these slots, for branch and floating-point operations, remain unused in both cycles shown. Instead of coalescing similar operations together as a SIMD operation, the operations can be fetched separately using the empty slots. In this way, the ISA does not need to be modified to support SIMD operations. At run-time, the processor would steer operations to an appropriate functional unit for execution as determined by the compiler. The hardware would know which functional unit to steer the



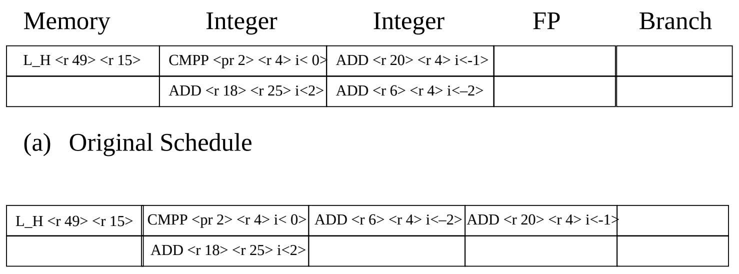
Memory	Integer	Integer	FP	Branch
L_H <r 49> <r 15>	CMPP <pr 2> <r 4> i< 0>	ADD <r 20> <r 4> i<-1>		
	ADD <r 18> <r 25> i<2>	ADD <r 6> <r 4> i<–2>		

(a) Original Schedule

L_H <r 49> <r 15>	CMPP <pr 2> <r 4> i< 0>	ADD <r 6> <r 4> i<–2> <r 20> <r 4> i<-1>		
	ADD <r 18> <r 25> i<2>			

(b) Schedule after homogeneous packing

Figure 6.3: Example of Homogeneous Operation Packing



Memory	Integer	Integer	FP	Branch
L_H <r 49> <r 15>	CMPP <pr 2> <r 4> i< 0>	ADD <r 20> <r 4> i<-1>		
	ADD <r 18> <r 25> i<2>	ADD <r 6> <r 4> i<–2>		

(a) Original Schedule

L_H <r 49> <r 15>	CMPP <pr 2> <r 4> i< 0>	ADD <r 6> <r 4> i<–2>	ADD <r 20> <r 4> i<-1>	
	ADD <r 18> <r 25> i<2>			

(b) Schedule after homogeneous packing using fetch bandwidth

Figure 6.4: Example of Homogeneous Packing exploiting Fetch Bandwidth

source operands to because the compiler can identify the ALU with each operation. In order to do so, the compiler annotates the ALU identifier to the operations at schedule time and the issue logic directs a fetched operation to this ALU. Using available fetch slots prevents an increase in register ports used per cycle, assuming a unified register file. Since the logic for steering operations manages the predictions at run-time, it is implemented as part of the predictor component and is elaborated in the next section.

Figure 6.4(b) shows the schedule of the previous example after packing operations using the FP slot. It is expected that adequate unused fetch bandwidth can be made available for fetching additional ALU operations. From the analysis of several benchmark programs, it was found that floating point and branch operations comprise less than 1% of dynamic instructions. This observation indicates that fetch bandwidth is severely under-utilized, and by scheduling narrow operations in unused slots, we can take better advantage of the available bandwidth.

In the example above, packing operations did not reduce the schedule length since one add operation had to be delayed by a cycle. One of the restrictions of packing operations in previous work is that only similar operations are allowed to execute together on a functional unit. A study by Scott et al. [67] observes that most of the benefits of packing

Memory	Integer	Integer	FP	Branch
L_H <r 49> <r 15>	CMPP <pr 2> <r 4> i< 0>	ADD <r 20> <r 4> i<-1>		
	ADD <r 18> <r 25> i<2>	ADD <r 6> <r 4> i<–2>		

(a) Original Schedule

L_H <r 49><r 15>	CMPP <pr 2><r 4>i< 0>	ADD <r 20><r 4>i<-1>	ADD <r 6><r 4>i<–2>	ADD <r 18><r 25>i<2>

(b) Schedule after heterogeneous packing

Figure 6.5: Example of Heterogeneous Packing

homogeneous operations into SIMD-like instructions can be achieved by instruction-level parallelism on a machine with two or more ALUs. If heterogeneous operations are packed on an ALU, the performance savings would potentially be higher. Packing heterogeneous operations permits scheduling all of the operations in the example above in one cycle, as shown in Figure 6.5(b). In this example, the add and compare operations can be scheduled together because they use at most 16 bits. The compiler assigns the position within the ALU for each operation allowing the issue logic to steer these operations to an appropriate ALU sub-component.

Note that it is not possible to pack heterogeneous operations on an ALU using static SIMD synthesis because each combination of operations requires a separate SIMD opcode. For VLIW processors, the compiler can assign heterogeneous operations to the same ALU along with their placement positions within the ALU. For superscalars, performing heterogeneous packing at run-time incurs additional complexity to select the operations to be packed. If 8-bit and 16-bit heterogeneous operations were interleaved on a 32-bit ALU, a processor with a central issue window would require additional control logic to find and select the operations to pack on an ALU. Packing heterogeneous operations works well for VLIWs because we can statically schedule the operations to avoid the run-time overhead of detecting narrow width operations.

6.6 Predictor

Bit-width prediction was implemented statically and the predicted widths were not varied dynamically. Hence, run-time history need not be computed for the operations and the predictor component does not require the design of predictor tables. Instead, the predictions are encoded within the operations at compile-time. The bits *predicted_8* and *predicted_16*, that were introduced in the previous section, encode the predictions. The issues within the predictor component include how to handle the predictions at run-time by steering the operands from the fetch slots to the appropriate functional unit. Also, in context of heterogeneous packing, an ALU would require the capability to execute operations of different types concurrently. Both these issues are outlined below.

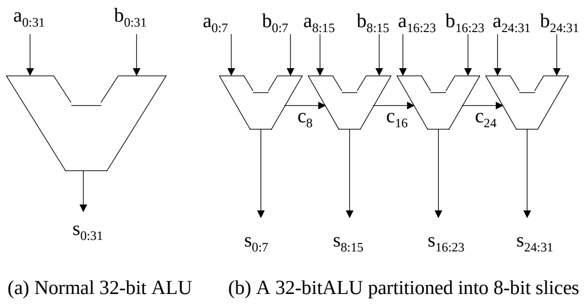


Figure 6.6: ALU structure

- *Partitioned ALU:* In order to pack heterogeneous operations together, an ALU needs to execute operations of different types in parallel. This functionality is provided by partitioning an ALU into sub-components executing different operations. It is possible to consider a 32-bit ALU as a group of 8-bit sub-ALUs executing the same operation and connected through carry lines. A pictorial view of a 32-bit ALU partitioned into sub-components is shown in Figure 6.6. Since an ALU is a combinational circuit of cascaded logic gates, this kind of partitioning can be made. Each sub-components is allowed to execute independently by providing a separate mode select to determine the type of operation to execute. Also, each sub-component was assigned its own overflow and zero bits. The work in this dissertation targets simple functional unit design applicable to embedded VLIW processors. Hence, these functional units can be extended without any major design modifications.

- *Steering logic*: The steering involves redirecting the source operand values to the ALU. This logic is not complex because the compiler statically assigns where to steer each operation and there are no dynamic steering decisions. Similar logic is already provided in several VLIW processors that have more functional units than fetch bandwidth, such as Starcore's SC140 and Trimedia's TM-1000 processor. Figure 6.7 shows a high-level view of the steering logic for a 5-wide VLIW architecture. The figure shows a 5-wide VLIW architecture with one memory, branch and floating-point unit, and two integer units. The integer units are shown as partitioned ALUs. The fetched operations are steered to the appropriate functional units by pre-decoding the resource bits associated with an operation. Operations that need to execute either on the memory, branch or floating-point unit are directly sent to the ALU from their re-

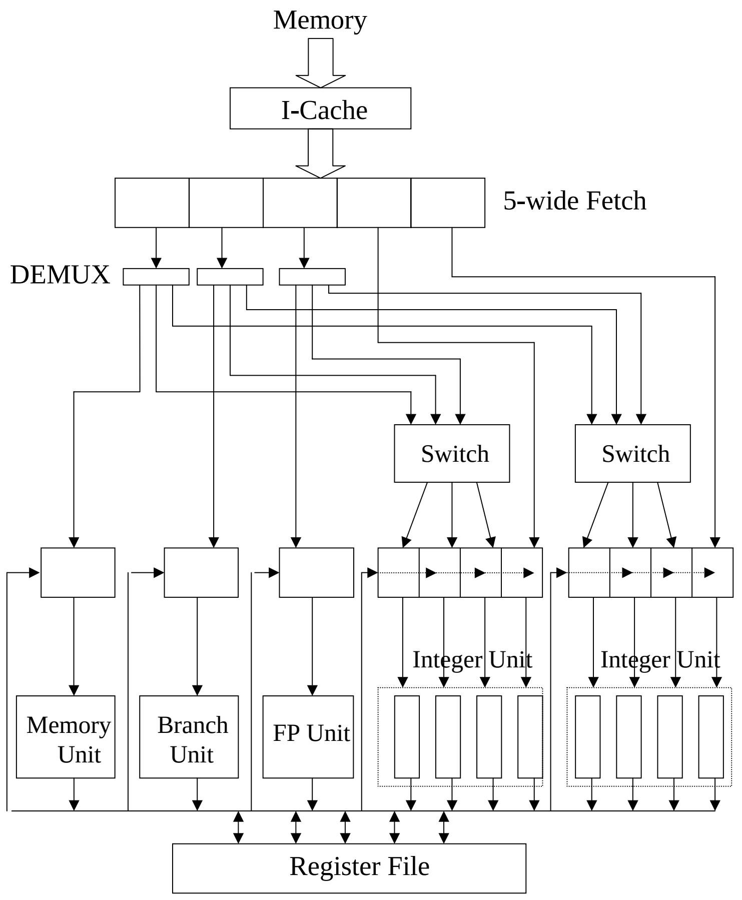


Figure 6.7: VLIW Architecture with Steering Logic

spective fetch slot. In case of ALU operations, the operation can be steered from the fetch slot of any functional unit, whenever multiple operations are packed together. For the example of Figure 6.5, the floating-point and branch slots are used to fetch the rightmost two add operations. The operations are routed through the switches as shown. These switches are necessary because an operation from a fetch slot may need to be routed to any sub-component of an ALU.

An observation from the figure 6.7 is that the number of register ports are not increased because of bit-width prediction. The potential candidates accessing the register file at the same time increases due to the capability of an ALU to execute multiple operations concurrently. However, the number of register accesses per cycle does not increase because the number of fetched operations per instruction word is preserved. It is assumed that each functional unit is associated with ports to the same register file. Hence, the floating-point and general purpose registers are clustered together in a unified register file. Also, the underlying architecture model of Trimaran allows the branch unit to access the register file. In case an integer unit needs to read several values because of packed operations, the register ports corresponding to some other functional unit are used for the reading. The steering involves sending the output value to the port that will write the value to the register file. Besides the functional unit identifier, the operations are also annotated with the register port used for reading/writing the values. If separate register files for integer and floating-point operations were present, the register ports would increase for general purpose registers. In that case, the increase in ports can be prevented by allocating all the values accessed by packed operations of an ALU to the same register. This idea will be elaborated while presenting results in the experiments section.

Although operand steering has some latency cost, the additional latency would not affect the machine cycle time. The reason is that typically register file accesses determine machine cycle time and other pipeline stages scale proportionate to the register file cycle time. Previous studies make this observation in context of superscalar processors and expect similar performance limits for VLIW processors [23]. The unused fraction of the cycle time in the fetch and execution stages can be used to perform the steering of the input operands and output value respectively.

6.7 Restorer

One of the observations in the example of Figure 6.5 is that scheduling using profile information about operand width decreases schedule length. Because this scheduling relies on profile information, the data widths of operands are being predicted and a mechanism to recover from mispredictions at run-time is required. The prediction threshold is used to control the aggressiveness of the predictions. By lowering the prediction threshold, more operations can be packed together. However, one of the drawbacks of lowering the threshold is that the misprediction rate increases. In case the operand width is more than its static prediction, the operation may not be computable with its assigned portion of the ALU in the same cycle. Such a misprediction needs to be identified at run-time and the processor should be able to correctly compute the values in those cases. Mispredictions can occur if either of the source operands requires more than the predicted bit-width or if the computed value spills over the allocated bits. The former case can be detected before the operation starts executing but the latter case is detectable only after execution has commenced.

For detecting mispredictions with the source operands, each register is annotated with two additional bits, namely *less_than_8* and *less_than_16*. As their name suggests, these bits specify whether the current value stored in that register is within 8 or 16 bits. A misprediction by an operation O due to one of its source operand accessing register R can occur by one of the following cases.

- $predicted\_8(O)$ AND $\neg less\_than\_8(R)$: This case is when the operation is predicted to have 8-bit operands and source register R has a value occupying more than 8-bits.
- $predicted\_16(O)$ AND $\neg less\_than\_16(R)$: This case corresponds to a misprediction of 16-bits

The mispredictions occurring due to the destination operand are identified at run-time using zero detection logic. This detection logic was proposed by Brooks et al. [8] for their dynamic packing scheme and already exists in several modern processors for detecting upper bytes to be zero. The logic consists of two signals, $zero\_16$ and $zero\_48$, that are associated with the upper 8 or 16 bits of the result of an ALU operation. These bits are set when the upper 16 bits or 48 bits of the computed value are zero. A misprediction associated with destination operand for an operation O occurs when either of the following is set:

- $predicted\_8(O)$ AND $\neg zero\_48(O)$ in case of 8-bit prediction or
- $predicted\_16(O)$ AND $\neg zero\_16(O)$ in case of 16-bit prediction.

The mechanism for recovery from data width mispredictions is to replay the mispredicted operation on an ALU. In this case, when an operand's width is mispredicted, the operation is replayed through the same ALU using the ALU's entire width during the following cycle. During the replay of a mispredicted operation, subsequent instructions are stalled in the pipeline. In the case of multiple mispredictions on a single ALU, each mispredicted operation is replayed independently and the pipeline is stalled during the entire replay sequence. The next section studies the impact of varying prediction threshold on misprediction rate and its effect on performance.

The ISA extensions used in the bit-width prediction framework are summarized below.

- Each operation is appended with two bits *predicted_8* and *predicted_16* which are set when the operation is predicted to have 8/16-bit operands.
- Each register operand is appended with the register port index used for accessing the operand value from the register file.

Also, the following hardware extensions are incorporated in the prediction framework.

- Each register is annotated with two bits *less_than_8* and *less_than_16* that are set when the register value is within 8/16 bits.
- Each ALU is partitioned into four sub-ALUs and a control signal is associated with each sub-component to determine the type of operation executing on the sub-ALU.
- Two control signals *zero_16* and *zero_48* are associated with the result of each sub-ALU operation and these bits are set when the upper 16 bits/48 bits of the computed value are zero.
- A switch is associated with each integer unit to steer operations from the fetch unit to their respective functional units.

6.8 Experimental Evaluation

6.8.1 Methodology

The performance of bit-width prediction was studied using the Trimaran compiler and simulator described in Chapter 5. The machine model used was a five-wide VLIW with two ALUs, one floating point unit, one memory unit, and one branch unit. A narrow model was used since bit-width prediction targets embedded VLIWs. Later in this section, the performance is evaluated across different machine widths. The 5-wide machine used for the experiments is similar to the Transmeta Crusoe TM5400 processor which is a four-wide VLIW with one ALU. However, the Crusoe processor can use its memory unit as an adder when no memory operations are scheduled, effectively making the processor five-wide. A fairer model of a five-wide VLIW was used that is less constrained by allowing two ALUs. The machine model has 64 general-purpose registers and a 32-bit data width. The analysis was performed after turning on the classical suite of optimizations along with aggressive code optimizations including dead code removal and redundancy elimination. The profile data included how often an operand width of an instruction was (1) less than 8 bits and (2) less than 16 bits. For example, the ALU assigned to the instruction was considered as 50% busy if profile data indicated an instruction to always have operands within 16 bits. The other half of the ALU was available to other instructions with narrow operands of 8 or 16 bits. The compiler used a prediction threshold of 100% for both 8-bit and 16-bit operands. Later, the prediction threshold is varied to study the effect of mispredictions on performance. In case an instruction's operand spilled over its predicted width, the processor stalls a cycle to re-execute the instruction.

6.8.2 Fetch Bandwidth and Register Pressure

For the initial studies, the potential performance of packing operations without bandwidth limitations was analyzed. Hence it was assumed there was always enough bandwidth to supply operations to fully pack an ALU. Also operations packed on an ALU were allowed to be of different types. Later in the chapter, these assumptions are removed. Figure 6.8 shows the percentage speedup for packing operations. The first column in the figure assumes perfect register allocation with no register spills and the second column incorporates register spills. The savings in absence of register spills are in the range 1-13% with an average of 5.7%. Another observation is that register spills become important and affect performance considerably. Since bit-width prediction increases parallelism, register pressure also increases. This reduction is especially noticeable in case of *ijpeg*. Speedups for the benchmarks after considering register spills are in the range 1-8% with an average of 2.6%. The impact of spills is particularly significant in the machine model used for experiments because there is only one load/store unit.

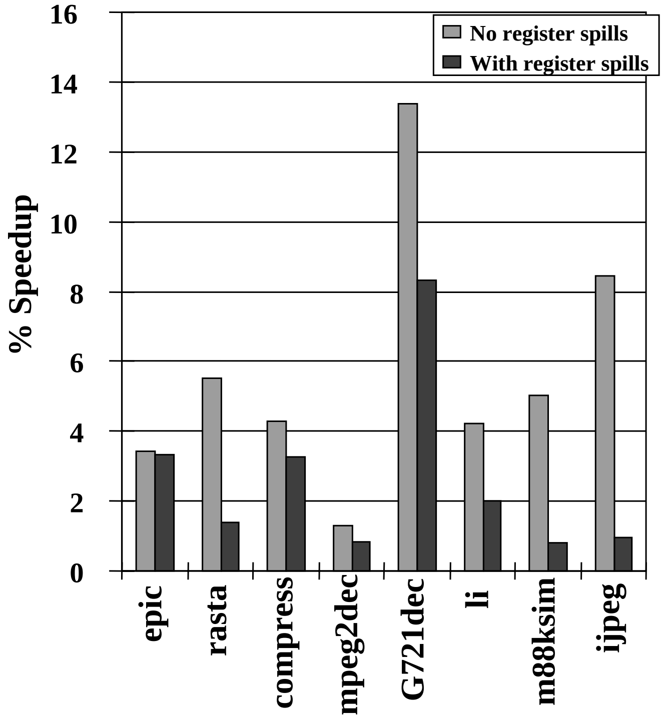
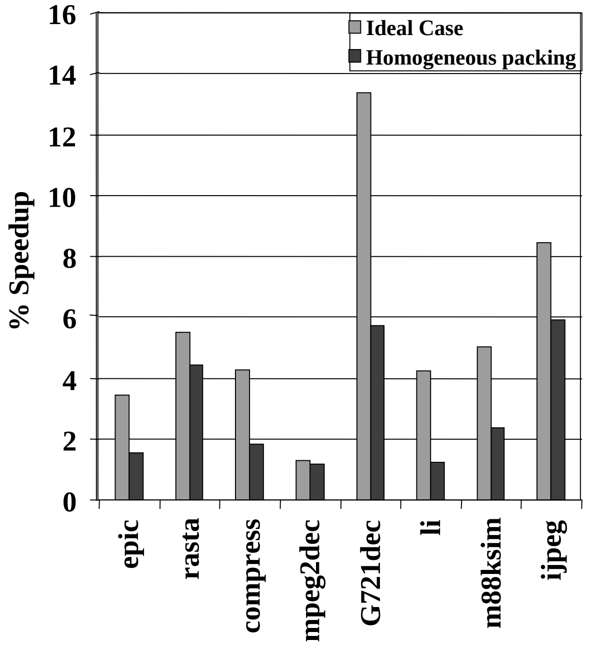


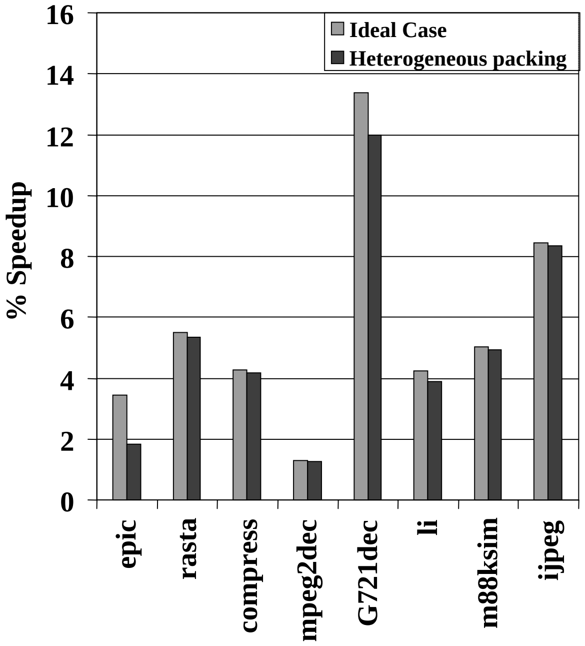
Figure 6.8: Performance with no limitation on Fetch Bandwidth

From the above observations, it can be inferred that in order to gain performance benefits from bit-width prediction, the register allocator needs to be sensitive to operand widths. One intuition is that 16-bit values can be spilled in consecutive locations and fetched in a single load operation. In the current implementation, bit-width prediction only packs ALU operations, so spills and reloads of narrow width data are performed with separate store and loads. By packing narrow operands for spills, the impact of spilling is expected to reduce. This packing is similar to SIMD synthesis of load/stores [42]. However, since register allocation is a static technique, we can also modify the allocation phase to accommodate spills of 16-bit values to the same register, which will significantly reduce the number of spills. This width-sensitive register allocation can be incorporated within a compiler independent of the underlying architecture. The analysis of the prediction framework in this dissertation involves evaluation of the scope of bit-width prediction for performance improvement. To avoid the impact of spills on this performance study, perfect register allocation with no spills was assumed for the rest of the experiments.

6.8.3 Homogeneous vs. Heterogeneous Packing



(a) Homogeneous packing



(b) Heterogeneous packing

Figure 6.9: Performance for limited Fetch Bandwidth

The next set of experiments analyze how constraining bandwidth affects performance. Hence the fetch bandwidth was fixed to be the width of the machine. Also, a study of the performance of packing heterogeneous operations over SIMD-like synthesis was performed. Two sets of experiments were carried out. First, only homogeneous operations were packed on an ALU. This technique is similar to the synthesis of SIMD instructions. In the second set, heterogeneous operations were allowed to be packed on an ALU. Figure 6.9 shows the performance for both experiments. Each case is compared with the performance of the case with no limitation on fetch bandwidth (labeled as ideal case). From Figure 6.9(a), it is observed that speedups in the case of packing homogeneous operations are considerably reduced from the ideal case. This reduction relates to the fact that fetch bandwidth is limited and only similar operations are allowed to share an ALU. For the case of heterogeneous packing, Figure 6.9(b) shows that performance improvements are close to the ideal case. This observation implies that heterogeneous packing uncovers significant parallelism that is not exploitable with homogeneous packing. Also, in the latter case, the unused fetch bandwidth is used to exploit most of the possible benefits of the ideal case especially for *G721dec, m88ksim, li* and *compress*.

6.8.4 Prediction Threshold and Misprediction

As discussed previously, prediction threshold can have an impact on detecting opportunities to pack operations. To determine whether this was the case, we varied the threshold and measured the impact on performance. It was found that as the prediction threshold was lowered, the schedule length improved but the misprediction rate also increased. It was observed that the improvements in schedule length were minor unless the prediction threshold was below 60%. This observation correlates to the previous study in operation packing [8] that notes that there are minor fluctuations in operand precision from less than 16-bit to greater 16-bit. For prediction thresholds lower than 60%, the cost of recovering from mispredictions mitigated any improvement in schedule length. With the current recovery mechanism, each mispredicted operation is replayed independently and contributes a cycle to the misprediction penalty. In case of complex arithmetic operations with latency of several cycles, this penalty can be more than one cycle. In the experiments, it was observed that less than 1% of packed operations had latency higher than one cycle (except in case of *G721dec* with 3.3%). Accounting for multi-cycle misprediction penalty would have a negligible effect on performance.

It is possible to achieve higher speedups with lower prediction thresholds by reducing the misprediction penalty. One way to reduce the penalty is by recovering multiple predictions in parallel with execution of the next VLIW instruction, if resources are available. Also more sophisticated prediction techniques may be able to predict more accurately reducing the instances of misprediction.

6.8.5 Machine Width

In the next experiment, the machine width was varied to determine in what context bit-width prediction is most appropriate. Three different machine models were considered: (1) a four-wide VLIW with one load/store, one ALU, one floating-point and one branch unit, (2) a five-wide VLIW with one load/store, two ALU, one floating-point and one branch unit and (3) an eight-wide VLIW with two load/store, four ALUs, one floating-point and one branch unit. These machine models were selected based on a range of embedded processors from a simpler model (four-wide Transmeta TM5400) to a relatively complex one (eight-wide TI TMS320C6201).

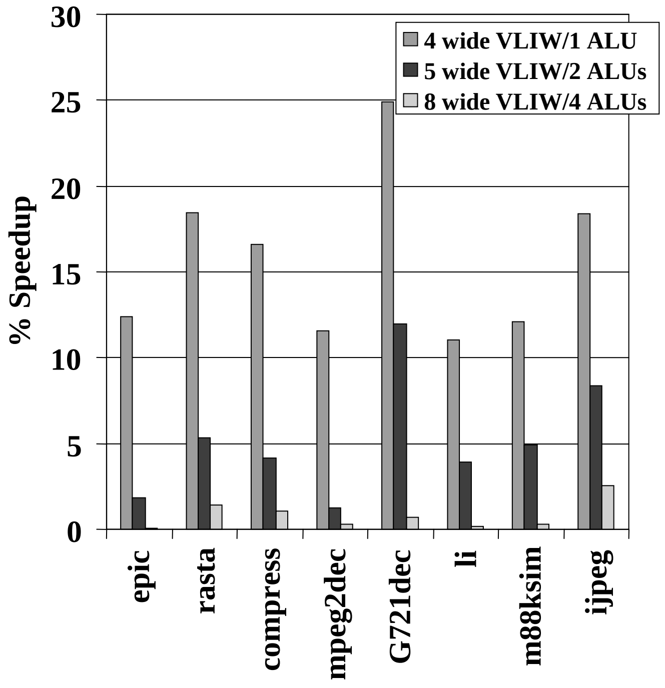


Figure 6.10: Performance for different Machine Models

Performance results for the different machine models is shown in Figure 6.10. From this figure, the narrow VLIW machine has considerable improvement, ranging from 11-25% with an average of 15.7%. However, as more ALUs are added to the machine model, these savings reduce. Performance improvements for the eight-wide machine model were insignificant, ranging from 0.1-3%. The reason is that this machine model has sufficient integer units to exploit most of the parallelism available by packing narrow operands. This observation confirms the statement that bit-width prediction is beneficial for resource-constrained VLIWs, similar to models 1 and 2. In the experiments, a 32-bit data width was used and a wider data path would allow more operations to be packed together on a functional unit.

From Figure 6.10, bit-width prediction substantially improves the performance for narrow VLIWs, and it is expected that the performance improvements would be even greater for VLIWs with a wider data width.

From the above results, it can be inferred that the prediction framework was able to effectively exploit operand widths for implementing bit-width prediction. The performance improvements due to heterogeneous packing show that packing operations of different types has much higher benefit than packing operations in a SIMD-like fashion. The bit-width optimization is most suitable for VLIW processors that are resource-constrained, such as those used in embedded systems. The experiments indicate that register allocation plays an important part in speculation of operand widths and the developed bit-width optimization can be used along with allocation of live ranges to registers based on the width of the live range value.

Chapter 7
Conclusions

The success of fine-grained parallelism in RISC processors has come from innovations in both compiler and architecture technologies. One particular domain of architectures that relies on fine-grained parallelism for performance is the VLIW architecture. Exploiting instruction-level parallelism in a VLIW processor has been traditionally quite challenging due to its static scheduling model which prevents instructions from being reordered dynamically. Scheduling at compile-time in a VLIW processor involves conservative dataflow analysis and is unable to exploit run-time information. Profiling has been used in this environment to design optimizations for a VLIW compiler that estimate run-time behavior of instructions. Among the optimizations useful for VLIWs is the class of prediction-based optimizations that reorder instructions at compile-time by predicting certain instruction attributes, such as computed values. Implementing prediction-based optimizations in a VLIW architecture through a compiler poses several challenges. The predicted instructions need to be selected at compile-time for scheduling. Also, a misprediction can incur substantial penalty because the instructions cannot be reordered dynamically to handle the mispredictions. Profile information is implicitly used in several VLIW compiler transformations that are prediction-driven. Examples of such transformations are region-formation and inlining. These transformations use profile information to predict the frequently executed basic blocks to optimize, and are typically applied at the source or basic block-level. There has been very limited prior work investigating fine-grained optimizations that explicitly predict instruction attributes in VLIW compilers. Most of these instruction-level prediction optimizations have been designed in isolation. A general infrastructure for prediction on a VLIW processor would facilitate translation of a novel prediction-driven optimization on the VLIW architecture and compiler.

7.1 Summary of contributions

This dissertation develops a framework useful for exploiting fine-grained parallelism in VLIW processors using prediction techniques. The framework provides a conceptual view of a prediction-driven compiler optimization for a VLIW architecture. The framework has been used within this dissertation to implement and evaluate specific optimizations. The optimizations are developed by applying the framework components in the domains of profiling, compilation and architecture. The framework is applied to develop methodologies for prediction and recovery by integrating concepts from the different domains in a synergistic way. The optimizations use feedback-directed compiler transformations and architectural design to exploit the transformations.

Among the optimizations developed in this dissertation is value prediction, which predicts computed values to allow dependent instructions to execute speculatively. The framework components were implemented for value prediction of load instructions because of their larger latency. Besides using conventional prediction schemes for predicting, this dissertation develops novel schemes that use the context of path history for prediction. The path along which an instruction gets executed is estimated and different paths are associated with different values for the instruction. The path-based techniques are developed by augmenting conventional prediction schemes, such as last value and stride prediction, with path information. The prediction accuracies are improved by 7-8% on average.

Within the value prediction optimization, the dissertation develops a dataflow-based algorithm that analyzes the effect of predicting an instruction on the critical path of the encompassing region. The developed analysis is able to filter 85% of the potential predictions that are not useful in improving the instruction schedule. The impact of this usefulness analysis is also observable in the value profiler as the overhead of profiling is reduced by more than 50% in all but one benchmark. As part of implementation of the prediction framework, this dissertation designs a hybrid predictor that is segmented into different partitions that store different amounts of prediction history. Based on the predictability behavior of an instruction, as observable from the profile data, an instruction is statically assigned into one of the predictor partitions. The smaller partition predicts instructions using simple prediction schemes and the larger partition uses hybrid prediction. The hybrid partition interleaves several hybrid configurations based on the most effective configuration for each instruction from profile data. Predicted instructions are statically assigned to table entries since the predictions are known at compile-time. Static assignment improves prediction accuracy, and the predictor designed in this dissertation has an overall average prediction accuracy of 77%. Both static and dynamic recovery schemes for value prediction were developed and compared for performance. Static misprediction recovery creates a single recovery block for all the predictions within a region. The operations incorrectly speculated are re-executed from this recovery block using predication. Dynamic recovery is handled by designing an additional execution engine, Compensation Code Engine, that executes recovery code in parallel with the speculated code. Recovery code is generated at run-time and re-executed whenever the associated prediction is incorrect. The improvements in performance for the static recovery case range between 3-4% for most of the benchmarks. On the other hand, dynamic recovery improves performance in the range 5-7% for all except one benchmark. The improvements are higher in the dynamic case because the overhead of mispredictions is partially subsumed by executing in parallel with the speculated code. Also, the static case requires explicit code restructuring to create recovery blocks at compile-time which incurs additional latency. The advantage of static recovery is that the hardware does not need to be modified for incorporating value prediction.

Another optimization, bit-width prediction, was implemented using the prediction framework. This optimization predicts operand widths allowing ALU operations with narrow operands to be packed on the same ALU. The predictions are made statically to enable the compiler to pack the operations at compile-time. The instructions were profiled to observe how often the width of operands were within a byte or a half-word. Integer operations with narrow source and destination operands are packed together on the same ALU. This dissertation introduces the concept of heterogeneous packing in which operations of different types are packed together on the same ALU. The packing avoids increases in fetch bandwidth by using the unused fetch slots to fetch the packed operations and steering them to the appropriate ALU. Only operations with high probability of having narrow operands, from the profile data, are packed together. Predictor tables are not used since the bit-widths are predicted statically. Misprediction recovery is handled by stalling execution whenever an operation spills over its predicted operand size and resuming after re-executing the mispredicted operations. Heterogeneous packing outperforms SIMD-like packing significantly and improves performance by 1-13% and 5% on average. The improvements increase up to 15% on average as the width of the instruction word is decreased but are minimal for wide VLIW machine models. Hence, the bit-width prediction optimization is found to be more suitable for embedded VLIW architectures that are constrained in resources like functional units and registers.

The observations from implementing the above optimizations indicate the prediction framework to be quite effective in addressing the challenges of fine-grained parallelism in VLIW architectures. The next section presents future work inspired from the work in this dissertation.

7.2 Future work

7.2.1 Extending the framework's applicability

The prediction framework developed in this dissertation was applied to address two major challenges to exploiting parallelism in VLIW architectures, data dependencies and resource dependencies. Data dependencies were addressed by predicting values and overcoming the latency of instructions computing those values. Resource dependencies were overcome by predicting operations with narrow operands and packing such operations to share the same functional unit. The former case addressed the problem of insufficient parallelism within the program while the latter case was suitable in resource-constrained processors where the available parallelism exceeded the resources in the underlying machine model.

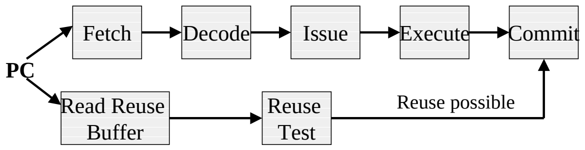


Figure 7.1: Instruction Reuse

An optimization that caters to both data and resource dependency within the same technique is *Instruction Reuse* [69]. This optimization was introduced in Chapter 2. It involves buffering previously computed results of an instruction and reusing the result value in future execution of the instruction whenever the input operands match. Figure 7.1 shows the pipeline stages after incorporating instruction reuse. The reuse buffer stores results of previously executed instructions. The reuse test queries this buffer to check whether a reusable result is available within the buffer. If so, the result is obtained from the buffer and execution of the instruction is bypassed. Instruction reuse increases the degree of parallelism by overcoming the latency of any operation whose result can be reused. This way, instructions dependent on the reused value can execute earlier than if no instruction reuse was applied. Also, instruction reuse improves the utilization of the machine resources by avoiding execution of instructions computing reused values. Hence, both program dependencies as well as resource dependencies can be handled by this optimization. The prediction framework can be used to develop the instruction reuse optimization for a VLIW architecture as follows.

- Usefulness Analyzer: The program regions where instruction-reuse would prove beneficial can be identified using dataflow analysis. This analysis can be built to identify entire extended blocks being used by the compiler for reusability. Another possibility is to build the reusable regions from program instructions using trace-building schemes and/or dataflow propagation techniques. Note that the useful regions would include regions devoid of parallelism as well as those regions constrained by resource dependencies upon scheduling.

- Selective Profiler: The regions found to be useful can be profiled to observe the re-usability for each instruction. An instruction that frequently computes values computed in a prior instance of its execution should be marked as reusable.

- Speculation Transformer: The instructions found to be reusable can be speculatively scheduled to read its value from the Reuse Buffer. By doing so, machine resources can be avoided for executing the instruction if the speculation is correct at run-time. Also, the latency of the reusable instruction would be overcome allowing the instructions dependent on it to be hoisted in the instruction schedule.

- Predictor: The values computed at run-time would be stored in the Reuse Buffer. Using some intelligent indexing schemes, this buffer can be accessed for each instruction predicted as being reusable to check and retrieve the result value.

- Restorer: In case an instruction is predicted as being reusable and its result value is not present in the Reuse Buffer, a misprediction occurs. Mispredictions can be handled by stalling the pipeline and re-executing the mispredicted instruction. Other possibilities include recovery blocks or specialized code that gets executed whenever there is a misprediction.

7.2.2 Future research directions

This dissertation has addressed the problem of fine-grained parallelism in VLIW architectures by providing a general infrastructure and developing novel optimizations using that infrastructure. There are several research directions that can be inspired from this work.

- **Integrate different granularity of predictions:** The dissertation work in value prediction focuses on instruction-level prediction. It is also possible to predict entire or partial blocks of code or even entire functions. These predictions are at coarser levels of granularity and involves interesting issues such as how to predict and recover entire regions. It is possible that some parts of a program that do not have predictable instructions consist of regions with predictable live-out values. One interesting aspect is to investigate the efficacy of predicting at different hierarchical levels of regions in the code.

- **Apply developed methodologies to other architectural platforms:** Some of the techniques resulting from the dissertation research can be applied across different architectures. For example, usefulness analysis in value prediction has been investigated in the domain of superscalar processors as a run-time technique [11]. Static usefulness analysis can be augmented as a filter to prediction in this domain, reducing the overall predictions. Also the segmented predictor design and selective profiling are useful concepts that can reduce prediction cost independent of the underlying architecture. A future extension to this dissertation research is to apply the techniques developed in this research to other architectural platforms.

- **Study interaction between optimizations:** One significant observation in the dissertation research is that an optimization cannot be designed independently of other phases of compilation. Although the prediction techniques increased instruction-level parallelism, they also increased register pressure resulting in larger number of spills. The additional spills subsume part of the benefits of the prediction. This phenomenon directly arises from the classical problem of interaction between scheduling and register allocation. Data prediction can be viewed as a pre-processing step of the scheduler to expose greater parallelism to the scheduler. There has been previous research integrating the phases of scheduling and register allocation and a new research direction is to view this integration from the perspective of data speculative code.

- **Bit-sensitive register allocation:** The design of width prediction also exposed the interaction between optimizations mentioned in the previous paragraph. Several operations that were packed together on a functional unit were found to have their values spilled to memory. An approach to reduce the effect of spilling narrow operand values is to assign live ranges of two or more such values to the same register. This bit-sensitive register allocation can be incorporated within a compiler independent of the underlying architecture.

- **Investigate other factors besides performance:** With the explosive growth of embedded processors over the past couple of years, processing capability is no longer the sole important issue in system design. Additional factors such as energy and power consumption are becoming primary design considerations for battery operated and portable embedded systems. It is challenging to apply existing optimizations and propose new optimizations in the embedded domain to enhance performance and reduce energy consumption. Recently, a software tool was developed, MPOWER, that measures the effectiveness of reordering memory accesses on bus switching activity [14]. The evaluations of MPOWER indicate encouraging potential for reducing switching activity and serve as a motivation to conduct further research in the area of power reduction. The synergistic ideology shall prove beneficial in designing novel optimizations addressing power consumption without severely affecting performance.

Bibliography

Bibliography

[1] The Trimaran compiler research infrastructure. *Tutorial Notes*, November 1997.

[2] A. Aiken and A. Nicolau. A development environment for horizontal microcode. *IEEE Transactions on Software Engineering*, May 1988.

[3] D. August, B. Deitrich, and S. Mahlke. Sentinel scheduling with recovery blocks. *Technical Report CRHC-95-05, Center for Reliable and High-Performance Computing, University of Illinois*, February 1995.

[4] T. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. *Proceedings of the 32nd ACM/IEEE International Symposium on Microarchitecture*, November 1999.

[5] T. Ball and J. R. Larus. Efficient path profiling. *Proceedings of the 29th ACM/IEEE international symposium on Microarchitecture*, December 1996.

[6] B. Black, B. Mueller, S. Postal, R. Rakvie, N. Utamaphethai, and J. P. Shen. Load execution latency reduction. *Proceedings of the 12th ACM International Conference on Supercomputing*, June 1998.

[7] R. Bodik and R. Gupta. Partial dead code elimination using slicing transformations. *Proceedings of the 1997 ACM conference on Programming language design and implementation*, June 1997.

[8] David Brooks and Margaret Martonosi. Dynamically exploiting narrow width operands to improve processor power and performance. *Proceedings of the Fifth IEEE International Symposium on High-Performance Computer Architecture*, January 1999.

[9] M. Burtscher and B. G. Zorn. Exploring last n value prediction. *Proceedings of the International Conference on Parallel Architectures and Compiler Techniques*, October 1999.

[10] B. Calder, P. Feller, and A. Eustace. Value profiling. *Proceedings of the 30th ACM/IEEE International Symposium on Microarchitecture*, December 1997.

[11] B. Calder, G. Reinmann, and D. Tullsen. Selective value prediction. *Proceedings of the 26th ACM/IEEE International Symposium on Computer Architecture*, May 1999.

[12] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W. W. Hwu. Profile-guided automatic inline expansion for C programs. *Software Practice and Experience*, May 1992.

[13] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwa. IMPACT: An architectural framework for multiple-instruction-issue processors. *Proceedings of the 18th ACM/IEEE International Symposium on Computer Architecture*, May 1991.

[14] B. R. Childers and T. Nakra. Reordering memory transactions for reduced power consumption. *Proceedings of the ACM SIGPLAN 2000 Workshop on Languages, Compilers and Tools for Embedded Systems*, June 2000.

[15] G. Chryos and J. Emer. Memory dependence prediction using store sets. *Proceedings of the 25th ACM/IEEE International Symposium on Computer Architecture*, June 1998.

[16] R. F. Cmelik and D. Keppel. Shade: A fast instruction set simulator for execution profiling. *Technical Report TR-93-12, Sun Microsystems Laboratories*, July 1993.

[17] D. Connors, H. Hunter, B. C. Cheng, and W. W. Hwu. Compiler-directed dynamic computation reuse: Rationale and initial results. *Proceedings of the 32nd ACM/IEEE International Symposium on Microarchitecture*, November 1999.

[18] D. Connors, H. Hunter, B. C. Cheng, and W. W. Hwu. Hardware support for dynamic management of compiler-directed computation reuse. *ACM SIGPLAN Notices*, November 2000.

[19] B. L. Deitrich and W. W. Hwu. Speculative hedge: Regulating compile-time speculation against profile variations. *Proceedings of the 29th ACM/IEEE International Symposium on Microarchitecture*, December 1996.

[20] D. DeVries. A vectorizing SUIF compiler: Implementation and performance. *Master's Thesis*, 1997.

[21] R. J. Eickemeyer and S. Vassiliadis. A load instruction unit for pipelined processors. *IBM Journal of Research and Development*, July 1993.

[22] J. R. Ellis. Bulldog: A compiler for VLIW architectures. 1986.

[23] K. I. Farkas, N. P. Jouppi, and P. Chow. Register file design considerations in dynamically scheduled processors. *WRL Research Report 95/10, Western Research Laboratory, Compaq*, 1995.

[24] J. Fisher. Trace scheduling: a technique for global microcode compaction. *IEEE Transactions on Computers*, July 1981.

[25] J. A. Fisher and B. R. Rau. Instruction-level parallel processing. *Science*, September 1991.

[26] C. Fu and T. M. Conte. Value speculation mechanisms for EPIC architectures. *Technical Report, Dept. of Electrical and Computer Engineering, North Carolina State University*, October 1998.

[27] C. Fu, M. D. Jennings, S. Y. Larin, and T. M. Conte. Software-only value speculation scheduling. *Technical Report, Dept. of Electrical and Computer Engineering, North Carolina State University*, June 1998.

[28] C. Fu, M. D. Jennings, S. Y. Larin, and T. M. Conte. Value speculation scheduling for high performance processors. *Proceedings of the 8th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[29] F. Gabbay and A. Mendelson. Speculative execution based on value prediction. *Technical Report TR#1080, EE Department, Technion - Israel Institute of Technology*, November 1996.

[30] F. Gabbay and A. Mendelson. Can program profiling support value prediction? *Proceedings of the 30th ACM/IEEE International Symposium on Microarchitecture*, December 1997.

[31] F. Gabbay and A. Mendelson. The effect of instruction fetch bandwidth on value prediction. *Proceedings of the 25th International Symposium on Computer Architecture*, December 1998.

[32] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. *Proceedings of the 6th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[33] A. Gonzalez, J. Tubella, and C. Molina. Trace-level reuse. *Proceedings of the 28th International Conference on Parallel Processing*, September 1999.

[34] J. Gonzalez and A. Gonzalez. The potential of data value speculation to boost ilp. *Proceedings of the 12th ACM International Conference on Supercomputing*, July 1998.

[35] R. Gupta. A code motion framework for global instruction scheduling. *Proceedings of the 7th International Conference on Compiler Construction*, March 1998.

[36] R. Gupta and M. L. Soffa. Region scheduling: An approach for detecting and redistributing parallelism. *IEEE Transactions on Software Engineering*, April 1990.

[37] J. Hennessey and D. Patterson. Computer architecture a quantitative approach. 1990.

[38] J. Huang and D. J. Lilja. Exploiting basic block value locality. *Proceedings of the 5th IEEE International Symposium on High-Performance Computer Architecture*, January 1999.

[39] P. S. Huang. Selecting better-performing alternative code using run-time profiling feedback. *M.S. Thesis, Department of Electrical Engineering and Computer Science, MIT*, June 1995.

[40] W. Hwu, S. A. Mahlke, W. Chen, P. Chang, N. Warter, R. Bringmann, R. Ouellette, R. Hank, T. Kiyohara, G. Haag, J. Holm, and D. Lavery. The superblock: An effective technique for VLIW and Superscalar compilation. *The Journal of Supercomputing*, May 1993.

[41] V. Kathail, M. Schlansker, and B. R. Rau. HPL playdoh architecture specification: Version 1.1. *Hewlett-Packard Laboratories Technical Report HPL-93-80(R.1), Computer Systems Laboratory*, February 2000.

[42] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. *Proceedings of the ACM 2000 Conference on Programming Language Design and Implementation*, June 2000.

[43] E. Larson and T. Austin. Compiler controlled value prediction using branch predictor based confidence. *Proceedings of the 33rd ACM/IEEE International Symposium on Microarchitecture*, December 2000.

[44] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communication systems. *Proceedings of the 30th ACM/IEEE International Symposium on Microarchitecture*, December 1997.

[45] C. G. Lee and D. J. DeVries. Initial results on the performance and cost of vector microprocessors. *Proceedings of the 30th ACM/IEEE International Symposium on Microarchitecture*, December 1997.

[46] S. J. Lee, Y. Wang, and P. C. Yew. Decoupled value prediction on trace processors. *Proceedings of the 6th IEEE International Symposium on High-Performance Computer Architecture*, January 2000.

[47] M. H. Lipasti. Value locality and speculative execution. *Ph.D. Thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University*, May 1997.

[48] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. *Proceedings of the 29th ACM/IEEE International Symposium on Microarchitecture*, December 1996.

[49] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. *Proceedings of the 7th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[50] S. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann. Effective compiler support for predicated execution using the hyperblock. *Proceedings of the 25th ACM/IEEE International Symposium on Microarchitecture*, December 1992.

[51] S. McFarling. Combining branch predictors. *Technicak Report DEC WRL TN-36*, June 1993.

[52] E. Morancho, J. M. Llaberia, and A. Olive. Split last-address predictor. *Proceedings of the International Conference on Parallel Architectures and Compiler Techniques*, October 1998.

[53] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. Dynamic speculation and synchronization of data dependences. *Proceedings of the 24th ACM/IEEE International Symposium on Computer Architecture*, June 1997.

[54] R. Muth, S. Watterson, and S. Debray. Code specialization based on value profiles. *Static Analysis Symposium*, June 2000.

[55] T. Nakra, B. R. Childers, and M. L. Soffa. Width-sensitive scheduling for resource-constrained VLIW processors. *Proceedings of the 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization*, December 2000.

[56] T. Nakra, R. Gupta, and M. L. Soffa. Global context-based value prediction. *Proceedings of the 5th IEEE International Symposium on High Performance Computer Architecture*, January 1999.

[57] T. Nakra, R. Gupta, and M. L. Soffa. Value prediction for VLIW machines. *Proceedings of the 26th ACM/IEEE International Symposium on Computer Architecture*, May 1999.

[58] G. Reinmann and B. Calder. Predictive techniques for aggressive load speculation. *Proceedings of the 31st ACM/IEEE International Symposium on Microarchitecture*, December 1998.

[59] B. Rychlik, J. Faistl, B. Krug, and J. P. Shen. Efficacy and performance impact of value prediction. *Proceedings of the International Conference on Parallel Architectures and Compiler Techniques*, October 1998.

[60] B. Rychlik, J. Faistl, B.P. Krug, A.Y. Kurland, J.J. Sung, M.N. Velev, and J. P. Shen. Efficient and accurate value prediction using dynamic classification. *Technical Report CMuART -98-01, Dept. of Electrical and Computer Engineering, Carnegie Mellon University*, 1998.

[61] T. Sato. Profile-based selection of load value and address predictors. *Proceedings of the International Symposium on High Performance Computing*, May 1999.

[62] T. Sato and I. Arita. Partial resolution in data value predictors. *Proceedings of 2000 International Conference on Parallel Processing*, August 2000.

[63] Y. Sazeides and J. E. Smith. Implementation of context-based value predictors. *Technical Report ECE-97-8, University of Wisconsin-Madison*, December 1997.

[64] Y. Sazeides and J. E. Smith. The predictability of data values. *Proceedings of the 30th ACM/IEEE International Symposium on Microarchitecture*, December 1997.

[65] Y. Sazeides and J. E. Smith. Modeling program predictability. *Proceedings of the 25th ACM/IEEE International Symposium on Computer Architecture*, June 1998.

[66] M. Schlansker and B. R. Rau. EPIC: An architecture for instruction-level parallel processors. *Hewlett-Packard Laboratories Technical Report HPL-1999-111, Computer Systems Laboratory*, February 2000.

[67] Kevin Scott and Jack Davidson. Exploring the limits of sub-word parallelism. *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, October 2000.

[68] M. Smith. Overcoming the challenges to feedback-directed optimization. *ACM SIGPLAN Notices*, July 2000.

[69] A. Sodani and G. S. Sohi. Dynamic instruction reuse. *Proceedings of the 24th ACM/IEEE International Symposium on Computer Architecture*, June 1997.

[70] A. Sodani and G. S. Sohi. An empirical analysis of instruction repetition. *Proceedings of the 8th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[71] A. Sodani and G. S. Sohi. Understanding the differences between value prediction and instruction reuse. *Proceedings of the 31st ACM/IEEE International Symposium on Microarchitecture*, November 1998.

[72] M. Stephenson, J. Babb, and S. Amarasinghe. Bitwidth analysis with application to silicon compilation. *Proceedings of the ACM 2000 Conference on Programming Language Design and Implementation*, June 2000.

[73] D. Tullsen and J. Seng. Storageless value prediction using prior register values. *Proceedings of the 26th ACM/IEEE International Symposium on Computer Architecture*, May 1999.

[74] D. M. Tullsen and B. Calder. Computing along the critical path. *Technical Report, University of California, San Diego*, 1998.

[75] G. Tyson and T. M. Austin. Improving the accuracy and performance of memory communication through renaming. *Proceedings of the 30th ACM/IEEE International Symposium on Microarchitecture*, December 1997.

[76] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. *Proceedings of the 30th ACM/IEEE International Symposium on Microarchitecture*, December 1997.

[77] N. J. Warter, S. A. Mahlke, W. W. Hwu, and B. R. Rau. Reverse if-conversion. *Proceedings of the ACM 1993 Conference on Programming Language Design and Implementation*, June 1993.

[78] S. Watterson and S. Debray. Goal-directed value profiling. *Proceedings of the 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization*, December 2000.

[79] T. Y. Yeh and Y. N. Patt. Alternate implementation of two-level adaptive branch prediction. *Proceedings of the 19th ACM/IEEE International Symposium on Computer Architecture*, May 1992.

[80] C. Young and M. D. Smith. Improving the accuracy of static branch prediction using branch correlation. *Proceedings of the 6th ACM International Conference on Architectural Support for Programming Languages*, October 1994.