ABSTRACT

FU, CHAO-YING

Compiler-Driven Value Speculation Scheduling.

(Under the direction of Prof. Thomas M. Conte.)

Modern microprocessors utilize several techniques for extracting instruction-level parallelism (ILP) to improve the performance. Current techniques employed in the microprocessor include register renaming to eliminate register anti- and output (false) dependences, branch prediction to overcome control dependences, and data disambiguation to resolve memory dependences. Techniques for value prediction and value speculation have been proposed to break register flow (true) dependences among operations, so that dependent operations can be speculatively executed without waiting for producer operations to finish. This thesis presents a new combined hardware and compiler synergy, *value speculation scheduling (VSS)*, to exploit the predictability of operations to improve the performance of microprocessors. The VSS scheme can be applied to dynamically-scheduled machines and statically-scheduled machines. To improve the techniques for value speculation, a value speculation model is proposed as solving an optimal edge selection problem in a data dependence graph. Based on three properties observed from the optimal edge selection problem, an efficient algorithm is designed and serves as a new compilation phase of benefit analysis to know which dependences should be broken to obtain maximal benefits from value speculation. A pure software technique is also proposed, so that existing microprocessors can employ *software-only value speculation scheduling (SVSS)* without adding new value prediction

hardware and modifying processor pipelines. Hardware-based value profiling is investigated to collect highly predictable operations at run-time for reducing the overhead of program profiling and eliminating the need of profile training inputs.

COMPILER-DRIVEN VALUE SPECULATION SCHEDULING

by

CHAO-YING FU

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

COMPUTER ENGINEERING

Raleigh

2001

APPROVED BY:

Prof. Thomas M. Conte
Chair of Advisory Committee

Prof. Paul D. Franzon

Prof. Wentai Liu

Prof. Eric Rotenberg

Biography

Chao-ying Fu was born on June 5, 1973 in Taichung, TAIWAN. Upon receiving the Bachelor of Science degree in Electrical Engineering from National Taiwan University in June 1995, he began the graduate study at North Carolina State University, USA. In December 1996, he obtained the Master of Science degree in Computer Engineering. He then enrolled in the doctoral program in Computer Engineering under the supervision of Dr. Thomas M. Conte. He completed the Ph.D. degree in May 2001.

Acknowledgements

First of all, I thank God for directing my life and leading me to North Carolina State University. Next, I acknowledge the contribution of Prof. Thomas M. Conte, my Ph.D. advisor, to this thesis. He has led me to do research on compiler-driven value speculation scheduling and investigate a pure software technique for value speculation since the beginning of my Ph.D. study. Without his vision, this thesis cannot be finished. I feel fortunate to have worked under his guidance. Also, I would like to thank Prof. Thomas M. Conte, Prof. Paul D. Franzon, Prof. Wentai Liu, and Prof. Eric Rotenberg to be my thesis committee. Thanks to Prof. Albert J. Shih for being the graduate school representative.

I would like to thank Dr. Youfeng Wu and Dr. Allan D. Knies. I learned a lot from them when I was an intern in Intel in 1997 and 1999.

I would like to thank all previous and current members in the Tinker group; Kishore Menezes, Sumedh Sathaye, Sanjeev Banerjia, Bill Havanki, Sergei Larin, Matt Jennings, Emre Ozer, Mark Toburen, Kim Hazelwood, Vikram Rao, Tripura Ramesh, and Huiyang Zhou. With their help, the LEGO compiler can serve as the tool for this thesis.

I would like to thank my grandmother, Yu-Chu, my father, Jeng-Nan, my mother, Huei-Yang, my sisters, Wei-Li and Wei-Ting, and brothers-in-law, Kuo-Shu and Fu-I, for their abundant love and support.

Last, I would like to thank all my friends in Raleigh. Because of them, I had a wonderful student life at NC State.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

Modern microprocessors utilize several techniques for extracting instruction-level parallelism (ILP) to improve the performance. Current techniques include register renaming to eliminate register anti- and output (false) dependences, branch prediction to overcome control dependences, and data disambiguation to resolve memory dependences [1], [41]. Recent research focuses on using value prediction [2], [3], [4] to break register flow (true) dependences, so that dependent operations can be speculatively executed without waiting for producer operations to finish. In this thesis, the technique for allowing speculative execution based on value prediction [6] is called *value speculation* [22].

The previous techniques for value speculation utilize hardware-only mechanisms [2], [3]. In these schemes, the instruction address (PC) of a register-writing instruction is sent to a value predictor to index a prediction table at the beginning of the fetch stage.

During the fetch and dispatch stages, the value predictor generates a prediction that is forwarded to a dependent instruction prior to its execution stage. The value speculative dependent instruction must remain in a reservation station (even while its own execution continues), and be prevented from retiring. At the state-update stage, the predicted value is compared with the actual result. If the prediction is correct, the dependent instruction can then release the reservation station, update system states, and retire. If the predicted value is incorrect, the dependent instruction needs to be re-executed with the correct operand. Figure 1 illustrates the pipeline stages for value speculation utilizing a hardware-only scheme.

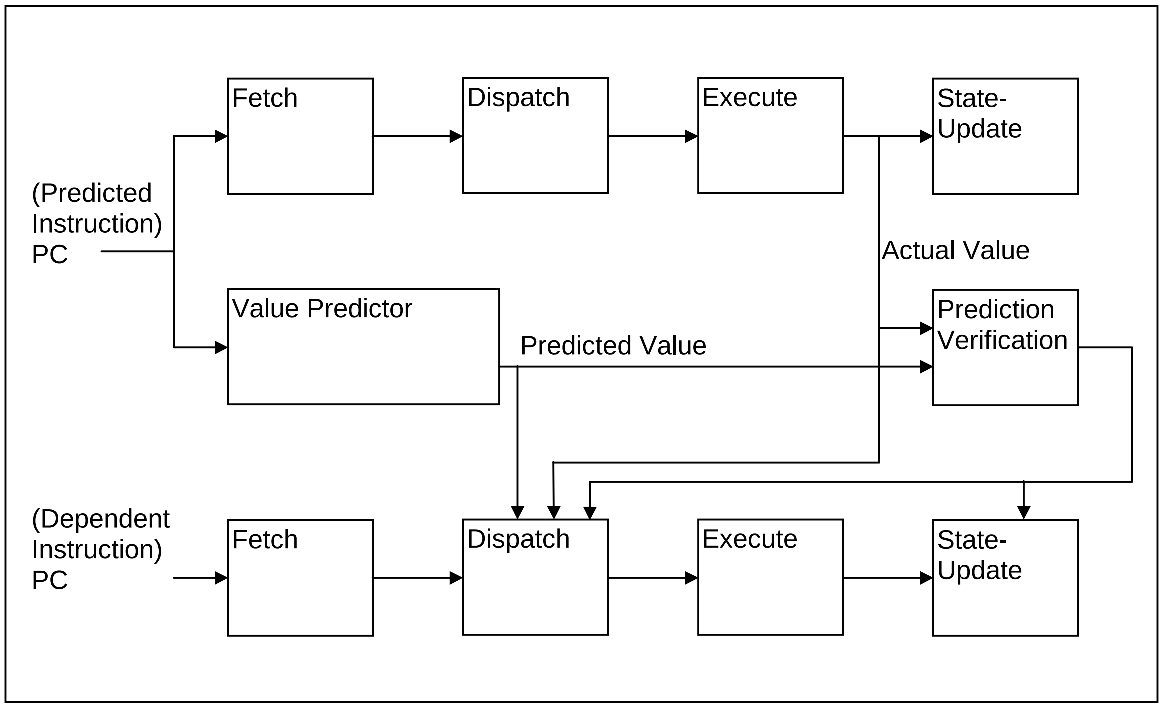


Figure 1.1 Pipeline stages of the hardware-only value speculation mechanism for flow dependent instructions. The dependent instruction is speculatively executed at the same cycle as its producer instruction.

The hardware-only value speculation schemes shown in Figure 1.1 are suitable for dynamically-scheduled machines, such as superscalars, but they cannot be applied to statically-scheduled machines, including VLIW [20] and EPIC [27], [28] architectures. In a related approach to a different problem, the memory conflict buffer [1] was presented to dynamically disambiguate memory dependences. This allows the compiler to speculatively schedule memory references above other, possibly dependent, memory instructions. Recovery code, generated by the compiler, ensures correct program execution even when the memory dependences actually occur. Aggressively scheduling memory references that are highly likely to be independent of each other improves performance. Likewise, value-speculative scheduling attempts to improve performance by aggressively scheduling flow dependences that are highly likely to be eliminated through value prediction. Recovery code can also be used when values are mispredicted.

This thesis applies the memory conflict buffer scheme to value speculation and proposes a new combined hardware and compiler synergy, which is called *value speculation scheduling (VSS)*. Two new predicting and updating operations, LDPRED and UDPRED, are proposed to be the interface between the value predictor and program code. Static VLIW instruction scheduling techniques are used to speculate value dependent operations aggressively. Hardware value predictors can provide predicted values for allowing the execution of speculated operations to continue. In the case of value misprediction, control flow is redirected to recovery code so that the execution can proceed with correct results. The VSS techniques leverage advantages of both hardware schemes for value prediction and compiler schemes for exposing ILP. VSS can be though of as a static ILP transformation that relies on dynamic value prediction hardware. Several advantages of the VSS scheme are as follows.

- Static scheduling provides a larger scheduling scope for exploiting ILP transformations, identifying long dependence chains suitable for value prediction, and then re-ordering code aggressively.
- Value-speculative dependent operations can be executed as early as possible before the predicted operations that they depend on.
- The compiler controls the number of predicted values and assigns different indices to them for accessing the value prediction table. Only operations that the compiler deems are good candidates for predictions are then predicted, reducing conflicts for the hardware.
- Recovery code is automatically generated, reducing the need for elaborate hardware recovery techniques.
- Instead of relying on statically predicted values (e.g., from profile data), LDPRED and UDPRED operations access dynamic prediction hardware for enhanced prediction accuracy.
- VSS can be applied to dynamically-scheduled (superscalar) processors, statically-scheduled (VLIW) processors, or explicitly parallel instruction computing (EPIC) processors [27], [28].
- The non-intrusive design for the VSS scheme makes it easy to employ value prediction and value speculation in future microprocessors.

To improve the techniques for value speculation, a value speculation model is proposed as solving an optimal edge selection problem in a data dependence graph. Based on three properties observed from the optimal edge selection problem, an algorithm is designed to solve the optimal edge selection problem efficiently. Running the optimal edge selection algorithm finds an optimal set of edges (dependences) and the corresponding maximal benefit from value speculation. Examining the selected dependences provides insights into the instruction selection techniques that relate to the success of utilizing value speculation to improve the performance of microprocessors. Also, the optimal edge selection algorithm serves as a new compilation phase of benefit analysis to expose selected dependences to dynamically-scheduled and statically-scheduled machines. The compiler-directed edge selection can alleviate the burden for the hardware to decide which dependences should be broken at run-time.

Software-only value speculation scheduling (SVSS) is proposed and can be applied to existing microprocessors for improving the performance. The SVSS scheme utilizes software static stride value predictors to generate value predictions, so that dependent operations can be value-speculatively executed. The experimental results show that the performance of the software static stride value predictor is comparable to that of the hardware stride two-delta value predictor [10], [13]. Significant speedups are shown for applying SVSS to the SPECint95 benchmarks.

To reduce the overhead of program profiling and eliminate the need of profile training inputs, hardware-based value profiling is investigated to collect highly predictable operations at run-time. The value predictor with additional tag and counter fields is proposed as the scheme of hardware-based value profiling. At the retirement stage, operations access the value predictor and update the tag and counter fields. Upon context-switches or interrupts, the tags with the maximum saturating counter values are stored to the memory for recording highly predictable operations. From the experimental results, the proposed scheme of hardware-based value profiling can accurately identify highly predictable operations at run-time. The VSS optimization is experimented based on the feedback from hardware-based value profiling.

1.2 Research Contributions

The research contributions of this thesis are as follows.

- This thesis proposes value speculation scheduling (VSS) to exploit the value predictability of operations to improve the performance of microprocessors. The VSS technique leverages advantages of both hardware schemes for value prediction and compiler schemes for exposing ILP.
- Two new predicting and updating operations, LDPRED and UDPRED, are proposed to be the interface between the value predictor and program code.
- A value speculation scheduling algorithm is proposed to utilize LDPRED and UDPRED operations to break critical paths in a program to shorten execution time.
- A value speculation model is built as solving an optimal edge selection problem in a data dependence graph to understand and improve the techniques for value speculation.
- Three properties are observed from the optimal edge selection problem and help to design an efficient optimal edge selection algorithm.
- Running the optimal edge selection algorithm serves as a new compilation phase of benefit analysis to know how many optimization opportunities for value speculation exist in a program and find an optimal set of edges (dependences) to be broken via

value prediction. The selected dependences are then exposed to the hardware or the compiler to obtain maximal benefits from value speculation.

- Software-only value speculation scheduling (SVSS) is proposed and can be applied to existing microprocessors for improving the performance.
- Software static stride value predictors are designed to have comparable performance to hardware stride two-delta value predictors.
- Hardware-based value profiling is proposed to accurately collect highly predictable operations at run-time with fewer overheads.

1.3 Outline of the Thesis

The organization of this thesis is as follows. Chapter 2 presents the techniques for value speculation scheduling (VSS), including the microarchitectural support and the VSS algorithm. Chapter 3 introduces the value speculation model by formally presenting an optimal edge selection problem, and proposes an optimal edge selection algorithm. Chapter 4 describes compiler-directed edge selection to expose selected dependences to the hardware or the compiler. Chapter 5 proposes software-only value speculation scheduling (SVSS) that can be applied to existing microprocessors. Chapter 6 studies the profile shift and investigates hardware-based value profiling. Chapter 7 concludes this thesis and mentions future research directions.

Chapter 2

Value Speculation Scheduling

Research in value prediction shows a surprising amount of predictability for the values produced by register-writing operations [2], [3], [4], [6], [10], [13], [15], [16]. Several hardware-based schemes have been proposed to exploit this predictability by eliminating flow dependences for highly predictable operations [2], [3], [6], [8], [9]. Instead of using hardware-only mechanisms for value speculation (e.g., the scheme in Figure 1.1), this chapter introduces a combined hardware and compiler synergy that is called *value speculation scheduling (VSS).* Static VLIW instruction scheduling techniques are used to speculate value dependent operations by scheduling them above the operations whose results they depend on. Value prediction hardware is used to provide predicted values for allowing the execution of speculated operations to continue. In the case of mispredicted values, control flow is redirected to recovery code so that the execution can proceed with the correct results.

The remainder of this chapter is organized as follows. Section 2.1 presents the microarchitectural support for value speculation scheduling (VSS). Section 2.2 examines the value predictor design for VSS. Section 2.3 introduces the VSS algorithm. Section 2.4 presents experimental results and discusses the heuristics used in the VSS scheme. Section 2.5 concludes this chapter.

2.1 Microarchitectural Support for VSS

Hardware pipeline stages for the VSS scheme are shown in Figure 2.1. Two new predicting and updating operations, *LDPRED* and *UDPRED*, are introduced to be the interface with the value predictor during the execution stage. An LDPRED operation loads a predicted value generated by the value predictor into a specified general-purpose register. A UDPRED operation updates the value predictor with the actual result, resetting the device for future predictions after a misprediction. In Figure 2.1 of the VSS scheme, the microprocessor only needs to add a new value predictor and slightly modify the pipeline for accessing the value predictor at the execution stage. The non-intrusive design makes it easy to incorporate the VSS scheme into future microprocessors.

Figure 2.2 shows an example of using LDPRED and UDPRED operations to perform the VSS optimization. In the original code sequence of Figure 2.2(a), operations 1 to 6 form a long flow dependence chain, which must be executed sequentially. If the flow dependence from operation 3 to operation 4 is broken, via VSS, the dependence height of the resulting dependence chain is shortened. Furthermore, ILP is exposed by the resulting data dependence graph.

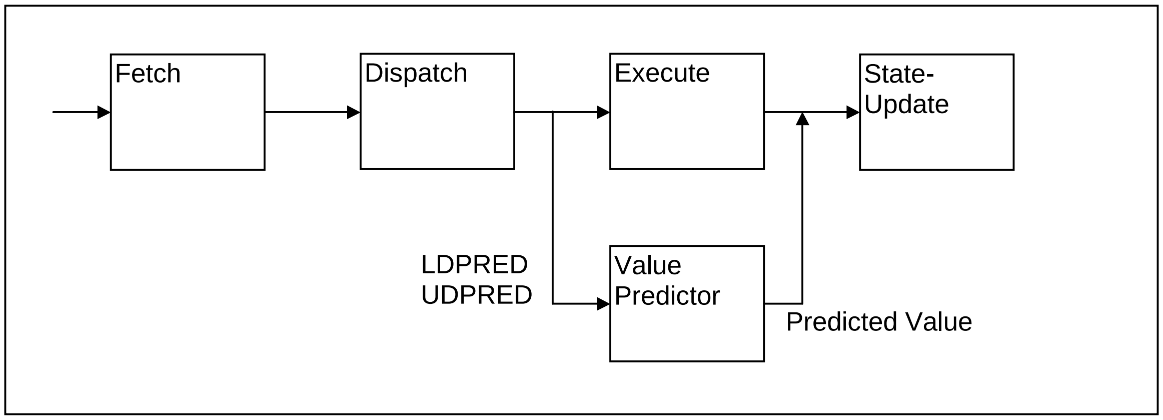


Figure 2.1 Pipeline stages of the VSS scheme. Two new operations, LDPRED and UDPRED, are introduced to be the interface with the value predictor during the execution stage.

(a) Original code

```
1:      ADD    R1 ← R2, 5
2:      SHL    R3 ← R1, 2
3:      LW     R4 ← 0(R3)
4:      ADD    R5 ← R4, 1
5:      OR     R6 ← R5, R7
6:      SW     0(R3) ← R6
Next:   ......
```

(b) New code after value speculation of R4 (the result of operation 3)

```
1:      ADD            R1 ← R2, 5
2:      SHL            R3 ← R1, 2
3:      LW             R4 ← 0(R3)
// load prediction  from hardware into R8
7:      LDPRED         R8 ← index
4':     ADD            R5 ← R8, 1
5':     OR             R6 ← R5, R7
6':     SW             0(R3) ← R6
// verify prediction
8:      BNE Recovery   R8, R4
Next:   ......

Recovery:
// update hardware predictor with R4
9:      UDPRED         R4, index
4:      ADD            R5 ← R4, 1
5:      OR             R6 ← R5, R7
6:      SW             0(R3) ← R6
10:     JMP Next
```

Figure 2.2 An example of value speculation scheduling.

Figure 2.3 shows the data dependence graphs for the code sequence of Figure 2.2 before and after breaking the flow dependence from operation 3 to operation 4. Assume that the latencies of arithmetic, logical, branch, store, LDPRED, and UDPRED operations are 1 cycle, and that the latency of load operations is 2 cycles. Then, the schedule length of the original code sequence of Figure 2.3(a), operations 1 to 6, is seven cycles. By breaking the flow dependence from operation 3 to operation 4, VSS results in a schedule length of five cycles. Figure 2.3(b) illustrates the schedule now possible due to reduced overall dependence height and ILP exposed in the new data dependence graph. This improved schedule length, from seven cycles to five cycles, does not consider the penalty associated with value misprediction due to the required execution of recovery code. The impact of recovery code on performance will be discussed in Section 2.3.

In Figure 2.2(b), the value speculation scheduler breaks the flow dependence from operation 3 to operation 4. Operations 4, 5 and 6 now form a separate dependence chain, allowing their execution to be speculated during scheduling. They become operations 4', 5', and 6' respectively. An operand of operation 4' is modified from R4 to R8. Register R8 contains the value prediction for destination register R4 of the predicted operation 3. Operation 7, LDPRED, loads the value prediction for operation 3 into register R8. When the prediction is incorrect (R8≠R4), operation 9, UDPRED, updates the value predictor with the actual result of the predicted operation, from register R4. Note that the resulting UDPRED operation is part of recovery code and its execution is only required when a value is mispredicted. To ensure correct program execution, the compiler inserts the branch (BNE), operation 8, after the store, operation 6', to branch to recovery code when the predicted value does not equal the actual value. The recovery code contains a

UDPRED operation and the original dependent operations 4, 5, and 6. After executing recovery code, the program jumps to the next operation after operation 8 and execution proceeds as normal. Note that in Figure 2.2(b) operations 4’, 5’, and 6’ use speculative versions [41] of original operations 4, 5, and 6. If the store, operation 6, does not have the speculative version, the compiler must not destroy data values belonging to other memory locations, i.e. the memory address of the store must be non-speculative. As shown in Figure 2.2(b), for aggressive optimization, the compiler may allow the store, operation 6’, to save wrong data values to the memory location of 0(R3), which is non-speculative.

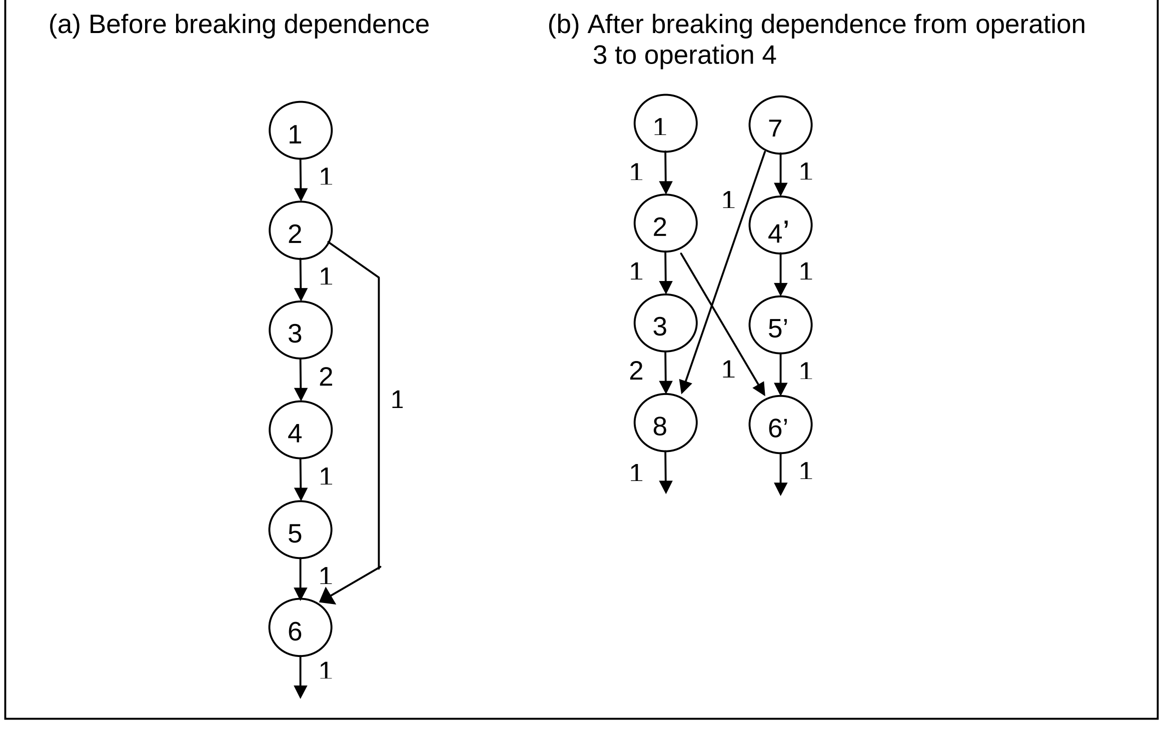

(a) Before breaking dependence
(b) After breaking dependence from operation 3 to operation 4
1
2
3
4
5
6
7
8
4’
5’
6’

Figure 2.3 Data dependence graphs for code in Figure 2.2. The numbers along each edge represent the latency of each operation. In (a), the schedule length is seven cycles. In (b), because of exposed ILP and dependence height reduction, the schedule length is reduced to five cycles.

Each LDPRED and UDPRED pair that corresponds to the same value prediction uses the same table entry index into the value predictor. Each index is assigned by the compiler to avoid unnecessary conflicts inside the value predictor. While the number of table entries is limited, possible conflicts are deterministic and can be factored into choosing which values to predict in a compiler approach. A value predictor design, featuring the new LDPRED and UDPRED operations, will be described in Section 2.2.

By combining hardware and compiler techniques, the strengths of both dynamic and static techniques for exploiting ILP can be leveraged. We see several possible advantages to VSS:

- Static scheduling provides a larger scheduling scope for exploiting ILP transformations, identifying long dependence chains suitable for value prediction, and then re-ordering code aggressively.
- Value-speculative dependent operations can be executed as early as possible before the predicted operations that they depend on.
- The compiler controls the number of predicted values and assigns different indices to them for accessing the prediction table. Only operations that the compiler deems are good candidates for predictions are then predicted, reducing conflicts for the hardware.
- Recovery code is automatically generated, reducing the need for elaborate hardware recovery techniques.
- Instead of relying on statically predicted values (e.g., from profile data), LDPRED and UDPRED operations access dynamic prediction hardware for enhanced prediction accuracy.

- VSS can be applied to dynamically-scheduled (superscalar) processors, statically-scheduled (VLIW) processors, or explicitly parallel instruction computing (EPIC) processors [27], [28].
- The non-intrusive design for the VSS scheme makes it easy to employ value prediction and value speculation in future microprocessors.

There is a drawback to the VSS scheme. Because static scheduling techniques are employed, value-speculative operations are committed to be speculative and therefore always require predicted values. Hardware-only schemes can dynamically decide when it is appropriate to speculatively execute operations. The dynamic decision is based on the value predictor's confidence in the predicted value, avoiding misprediction penalties for low confidence predictions.

2.2 Value Predictor Design

Microarchitectural support for value speculation scheduling (VSS) is in the form of special-purpose value predictor hardware. Value prediction accuracy directly relates to performance improvements for VSS. Various value predictors, such as last-value, stride, context-based, two-level, and hybrid predictors, provide different prediction accuracies [2], [3], [6], [10], [11], [13], [15], [16]. Value predictors with the most design complexity, in general, provide for the highest prediction accuracy. In order to feature LDPRED and UDPRED operations for VSS, previously proposed value predictors must be re-designed slightly.

Figure 2.4 shows the block diagram of a value predictor that includes LDPRED and UDPRED operations. In this value predictor, there are three fundamental units, the *current state* block, the *old state* block, and the *prediction hardware* block. The *current state* block may contain register values, finite state machines, history information, or machine flags, depending on the prediction method employed. The *old state* block hardware is a duplicate of the *current state* block hardware. The *prediction hardware* block generates predictions with the input from the *current state* block. Various prediction mechanisms can be used. For example, generating the prediction as the last value (last value predictors [2], [3]). Or, generating the prediction as the sum of the last value and the stride, which is the difference between the most recent last values (stride predictors [4], [6], [10], [13]). Also, two-level value predictors [13] and context-based value predictors [10], [11] allow for the prediction of recently computed values. For two-level predictors, a value history pattern indexes a pattern history table, which in turn is used to index a value prediction from recently computed values. Two-level value prediction hardware is based on two-level branch prediction hardware.

Both the LDPRED and UDPRED operations contain an immediate operand that specifies the value predictor table index. In general (independent of the prediction hardware chosen) the LDPRED operation performs three actions. The compiler assigned number indexes each action. First, the prediction hardware generates the predicted value by using the input from the current state block. Second, current state information is shifted to the old state block. Last, the current state block is updated based on the predicted value from the prediction hardware. Information used by the prediction hardware is updated simultaneously with the current state block update. Note that for the LDPRED operation, the predicted value is used to update the current state speculatively.

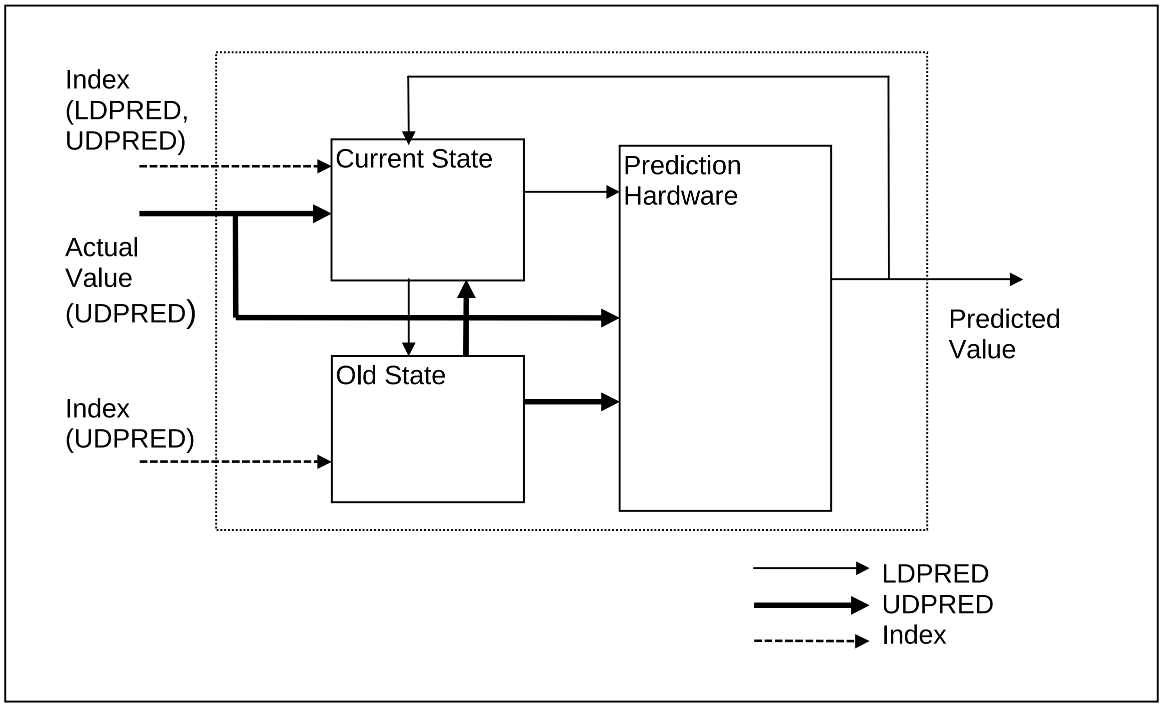


Figure 2.4 The block diagram of value predictor design featuring LDPRED and UDPRED operations.

The compiler assigned number also indexes the operation of the UDPRED operation. When the value prediction is incorrect, the recovery code in Figure 2.2(b) must be executed. The execution of UDPRED operations only occurs in recovery code, or only when values are mispredicted. The UDPRED operation causes the update of both the current state block and the prediction hardware with the actual computed value and the old state block.

If the compiler can ensure that each LDPRED and UDPRED pair is executed in turn (each prediction is verified and value predictions are not nested), the old state block requires only one table entry. The same table entry in the old state block is updated by every LDPRED operation, and used by every UDPRED operation, in the case of misprediction.

In the VSS scheme, a prediction needs to be generated for each LDPRED operation. There is no flag in the value predictor to indicate if a value prediction is valid or not. The goal of the value predictor is to generate as many correct predictions as possible. In Section 2.4, stride, two-level, and hybrid value predictors [13] are implemented to find the design, which provides the highest prediction accuracy for use in the VSS scheme. Stride predictors predict arrays and loop induction variables well. Two-level predictors capture the recurrence of recently used values and generate predictions based on previous patterns of values. However, neither of them alone can obtain high prediction accuracy for all programs, which exhibit different characteristics. Therefore, hybrid value predictors [13] consisting of both stride and two-level prediction are designed to cover both of these situations. Figure 2.5 shows such a hybrid predictor that obtains high prediction accuracy. The selection between the stride predictor and the two-level predictor is different from that in [13]. Every table entry has a saturating counter in the stride predictor and in the two-level predictor. The saturating counter increases when its corresponding prediction is correct, and decreases when its prediction is incorrect. Both saturating counters and predictors are updated for each prediction, regardless of which prediction is actually selected. The hybrid predictor selects the predictor with the maximum saturating counter value. In the event of a tie, the hybrid predictor favors the prediction from the two-level predictor. Prediction accuracy results for the three value predictors will be presented in Section 2.4.

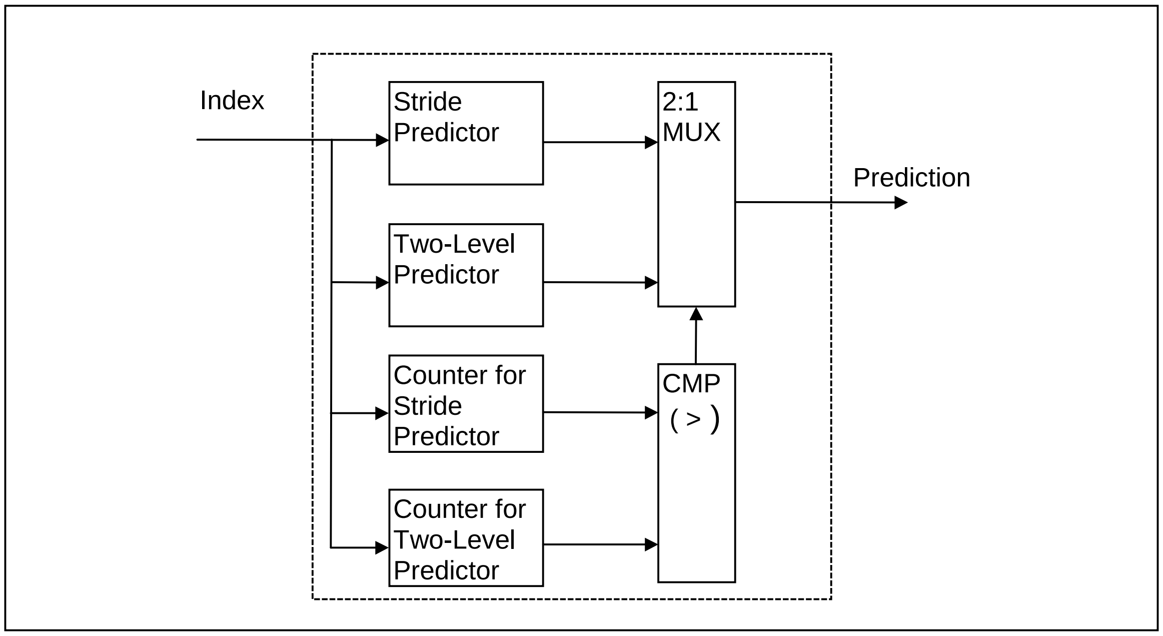


Figure 2.5 The hybrid predictor (with stride and two-level predictors). Saturating counters are compared to select between the prediction techniques.

2.3 A Value Speculation Scheduling Algorithm

Performance improvement for value speculation scheduling (VSS) is affected by prediction accuracy, the number of saved cycles (from schedule length reduction), and the number of penalty cycles (from execution of recovery code). Suppose that after breaking a flow dependence, value-speculative dependent operations are speculated, saving S cycles in overall schedule length when the prediction is correct. Recovery code is also generated and requires P cycles. Prediction accuracy for the speculated value is X. In this case, speedup may be positive if $S > (1-X) * P$ holds. For the example of Figure 2.3(b), VSS saves 2 cycles (from 7 cycles to 5 cycles) and the resulting recovery code contains 5 operations, requiring 3 cycles in an ILP processor. Therefore, for positive speedup, the prediction accuracy must be at least 33%. If the actual prediction accuracy is less, performance will be degraded by VSS. In Section 3.1, the penalties for value misprediction in the VSS scheme will be discussed in more detail. With these performance considerations in mind, an algorithm for VSS is proposed in Figure 2.6.

The first step is to perform value profiling. The scheduler must select highly predictable operations to improve performance through VSS. Results from value profiling under different inputs and parameters have been shown to be strongly correlated [4], [7]. Therefore, value profiling can be used to select highly predictable operations on which to perform value speculation.

Value profiling can be performed for all register-writing operations. If profiling overhead is a concern, a filter may be used to perform value profiling only on select operations. Select operations may be those that reside on critical paths (long dependence heights) or those that have long latencies (e.g., load operations). In [7], estimating and convergent profiling are proposed to reduce profiling overhead for determining the invariance of operations. Similar techniques could be applied for determining the value predictability of operations.

Next, the value speculation scheduler performs region formation. Treegion formation [17] is the region type chosen for our experiments. A treegion is a non-linear region that includes multiple execution paths in the form of a tree of basic blocks. The larger scheduling scope of treegions allows the scheduler to perform aggressive control speculation [41] and value speculation. A data dependence graph is then constructed for each region.

In step four, a prediction accuracy threshold is used to determine whether or not to perform value speculation on each operation. For each operation, the scheduler queries the value profiling information to get the estimate of its predictability. If the predictability estimate is greater than the threshold, value prediction is performed. For aggressive scheduling, more operations can be speculated by choosing a low threshold. Suggested values for the threshold are derived from experimental results in Section 2.4.

```
1.  Perform value profiling
2.  Perform region formation
3.  Build a data dependence graph for a region
4.  Select an operation with its prediction accuracy (based on value profiling) greater
    than a threshold
5.  Insert LDPRED after the predicted operation (the selected operation of step 4)
6.  Change the source operand of the dependent operation(s) to the destination register
    of LDPRED
7.  Insert a branch to recovery code
8.  Generate recovery code (which contains UDPRED)
9.  Repeat steps 4 – 8 until no more candidates found
10. Update the data dependence graph for a region
11. Perform instruction scheduling for a region
12. Repeat steps 2 – 11 for each region
```

Figure 2.6 A value speculation scheduling algorithm.

When an operation is selected for value prediction, an LDPRED operation is inserted directly after it. The LDPRED operation has an immediate value that is assigned by the scheduler to be its chosen index into the value predictor. A new register is also assigned as the destination of the LDPRED operation. Once the new destination register has been chosen for the LDPRED operation, any dependent operation(s) may update their source register(s) to reflect the new dependence on the LDPRED operation. Only the first dependent operation in a chain of dependent operations needs to update its register source, the remaining dependencies in the chain are unaffected. Even though more than one chain of dependent operations may result from just one value prediction, only one LDPRED operation is needed for each value prediction.

In step seven, a branch to recovery code is inserted for repairing value misprediction. Only one branch per data value prediction is required and the scheduler determines where this branch is inserted. Once the location of the branch is set, all operations in all dependence chains between the predicted operation and the branch to recovery code are candidates for value-speculative execution. It is therefore desirable to schedule any of these operations above the predicted operation. Actual hardware resources will restrict the ability to speculatively execute these candidates for value speculation. Also, as all candidates for value speculation are duplicated in recovery code, their number directly affects the penalty for value misprediction. These factors affect the scheduler's decision on where to place the branch to recovery code. Moreover, the compiler needs to make sure that all source operands (e.g., register values and memory data values) of candidate operations between the predicted operation and the branch are protected, so that inside recovery code value-speculative operations can be re-executed with original operands in the case of value misprediction.

In step eight, recovery code is created for repairing value misprediction. The recovery code contains the UDPRED operation, a copy of each candidate for value-speculative execution, and an unconditional jump back to the operation following the branch to recovery code. The UDPRED operation uses the same immediate value, assigned by the scheduler, as its corresponding LDPRED operation for indexing the value predictor. The other source operand for the UDPRED operation is the destination register of the predicted operation (the actual result of the predicted operation). The UDPRED operation index and the actual result are used to update the value predictor.

Finally, in steps ten and eleven, the data dependence graph is updated to reflect the changes and instruction scheduling for the region is performed. Because of the machine resource restrictions and dependences, not all candidates for value speculation are speculated above the predicted operation. Section 2.4 shows the results of using different threshold values for determining when to do value speculation.

2.4 Experimental Results

The SPECint95 benchmark suite was used in the experiments. All programs were compiled with classic optimizations by the IMPACT compiler from the University of Illinois [18] and converted to the Rebel textual intermediate representation by the Elcor compiler from Hewlett-Packard Laboratories [19]. Then, the LEGO compiler, a research compiler developed at North Carolina State University, was used to insert profiling code, form treegions, and schedule operations [17]. After instrumentation for value profiling, intermediate code from the LEGO compiler was converted to C code. Executing the resultant C code generated profiling data.

For the experiments in value speculation scheduling, load operations were filtered as targets for value speculation. Load operations were selected because they are usually on critical paths and have long latencies. Value profiling for load operations was performed on all programs. Table 2.1 shows the statistics from these profiling runs. The number of total profiled load operations represents the total number of load operations in each benchmark, as all load operations are instrumented (profiled). The number of static load operations represents the number of load operations that are actually executed. The difference between total profiled and static load operations is the number of load operations that are not visited. The number of dynamic load operations is the total of each load operation executed multiplied by its execution frequency.

Table 2.1 Statistics of total profiled, static and dynamic load operations.

SPECint95	Total Profiled Load Operations	Static Load Operations	Dynamic Load Operations
099.go	7,702	6,370	86,613,967
124.m88ksim	2,954	747	15,765,232
126.gcc	35,948	17,418	132,178,579
129.compress	96	72	4,070,431
130.li	1,202	414	24,325,835
132.ijpeg	5,104	1,543	118,560,271
134.perl	6,029	1,429	4,177,141
147.vortex	16,587	10,395	527,037,054

Stride, two-level, and hybrid value predictors were simulated during value profiling to evaluate prediction accuracy for each load operation. During value profiling, after every execution of a load operation, the simulated prediction is compared with the actual value to determine prediction accuracy. The value predictor simulators are updated with actual values, as they would be in hardware, to prepare for the prediction of the next use. Since the goal of value profiling is to measure the potential prediction accuracy of operations rather than the required capacities of the hardware buffers, no index conflicts between operations are modeled.

Each entry in the stride value predictor [4], [10], [13] has two fields, the *stride* and the *current value*. The prediction is always the current value plus the stride. The stride equals the difference between the most recent current values. The stride value predictor always generates a prediction. No finite state machine hardware is required to determine if a prediction should be used.

The two-level value predictor design is as in [13], with four data values and six outcome value history patterns in the value history table of the first level. The value history patterns index the pattern history table of the second level. The pattern history table employs four saturating counters, used to select the most likely prediction amongst the four data values. The saturating counters in the pattern history table increase by three, up to twelve, and decrease by one, down to zero. Selecting the data value with the maximum saturating counter value always generates a prediction.

The hybrid value predictor of stride and two-level value predictors utilizes the previous description illustrated earlier in Figure 2.5. In the hybrid design, the saturating counters, used to select between stride and two-level prediction, also increase by three, up to twelve, and decrease by one, down to zero.

Figure 2.7 shows the prediction accuracy of load operations under stride, two-level, and hybrid predictors. The prediction accuracy of the two-level predictor is higher than that of the stride predictor for all benchmarks except 129.compress and 132.ijpeg. However, the average prediction accuracy for the stride predictor is higher than that for the two-level predictor because of the large performance difference in 129.compress. Examining the value trace for 129.compress shows many long stride sequences that are not predicted correctly by the history-based two-level predictor. The hybrid predictor, capable of leveraging the advantages of each prediction method, has the highest prediction accuracy of 63% on an average across all benchmarks.

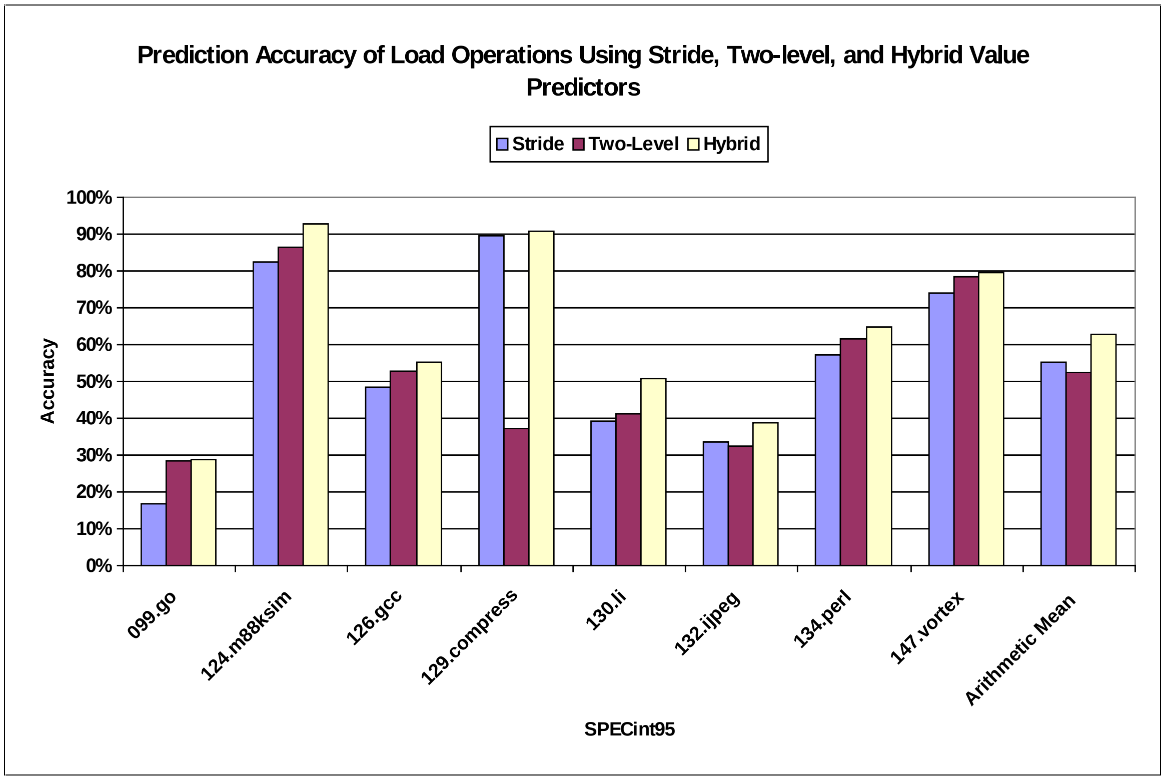


Figure 2.7 Prediction accuracies of load operations using stride, two-level, and hybrid predictors.

Figures 2.8 and 2.9 show the prediction accuracy distribution for load operations using the hybrid predictor. Figure 2.8 is the distribution for static load operations and Figure 2.9 is the distribution for dynamic load operations. For 124.m88ksim, 45% of the static load operations have prediction accuracies 90%, representing 90% of the dynamic load operations. For 129.compress, 70% of the static load operations have prediction accuracies of 90%, accounting for 80% of dynamic load operations. These static load operations are excellent candidates for VSS. Such high prediction accuracy results in low overhead due to the execution of recovery code. However, for benchmarks 099.go and 132.ijpeg respectively, only 15% and 25% of dynamic load operations have prediction accuracies above 50%. Therefore, they will not gain much performance from VSS.

The VSS algorithm shown in Figure 2.6 was performed on all SPECint95 programs. Prediction accuracy threshold values of 90%, 80%, 70%, 60% and 50% were evaluated. The number of candidates for value-speculative execution was limited to three for each value prediction. This parameter was varied in our evaluation, with the value of three providing good results.

For the evaluation of the speedup, a very long instruction word (VLIW) architecture machine model based on the Hewlett-Packard Laboratories HPL-PD architecture [20] was chosen. One cycle latencies are assumed for all operations (including LDPRED and UDPRED) except for load (two cycles), floating-point add (two cycles), floating-point subtract (two cycles), floating-point multiply (three cycles), and floating-point divide (three cycles). The LEGO compiler statically schedules the SPECint95 programs. The scheduler uses treegion formation [17] to increase the scheduling scope by including a tree-like structure of basic blocks in a single, non-linear region. The compiler performs control speculation [41], which allows operations to be scheduled above branches. Universal functional units that execute all operation types are assumed. An eight universal unit (8-U) machine model was used. All functional units are fully pipelined, with an integer latency of 1 cycle and a load latency of 2 cycles. Program execution time is measured by using the schedule length of each region and its execution profile weight. The effects of instruction cache and data cache are ignored, and perfect branch prediction is assumed in an effort to determine the maximum potential benefits of VSS.

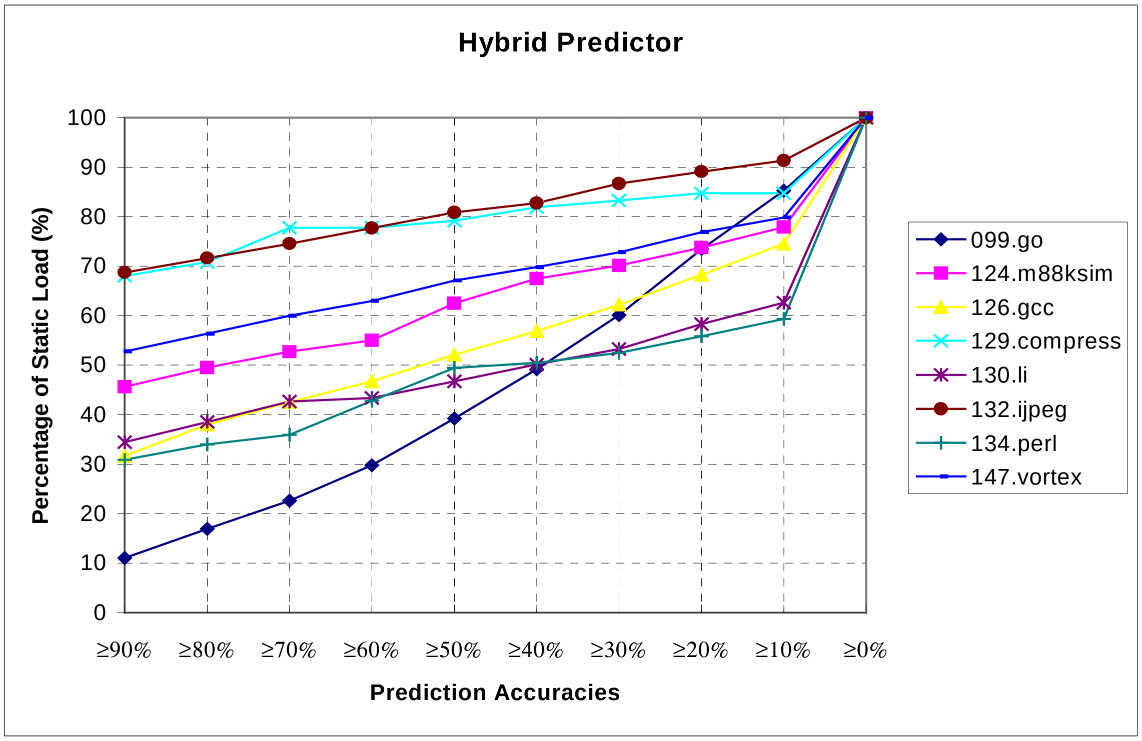


Figure 2.8 The prediction accuracy distribution for static load operations using the hybrid predictor.

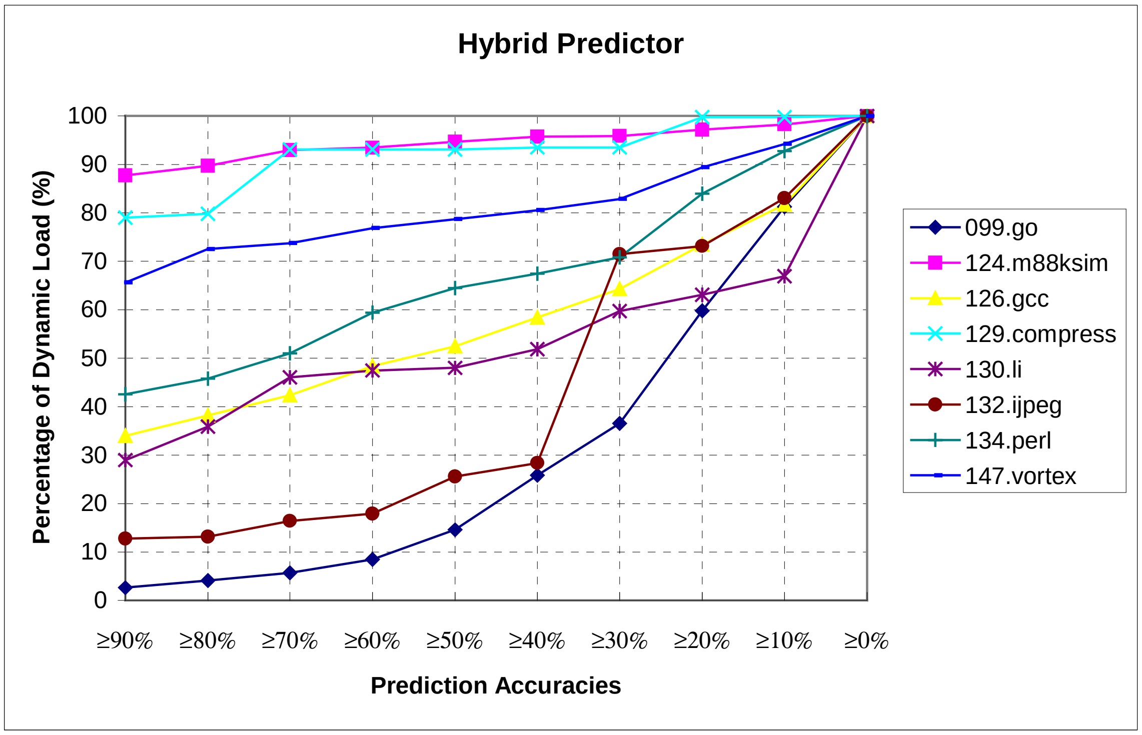


Figure 2.9 The prediction accuracy distribution for dynamic load operations using the hybrid predictor.

Figure 2.10 shows the execution time speedup of programs scheduled with VSS over without VSS. Five different prediction accuracy thresholds were used to select which load operations are value speculated. The maximum speedup for all benchmarks is 17% for 147.vortex. As illustrated in Figure 2.9, 147.vortex has many dynamic load operations that are highly predictable. While 147.vortex does not have the highest predictability for load operations, the sheer number, as illustrated in Table 2.1, results in the best performance. Benchmarks 124.m88ksim and 129.compress also show impressive speedups, 10% and 11.5% respectively, using a threshold of 50%. Speedup for 124.m88ksim actually goes up, even as the prediction accuracy threshold goes down, from 90% to 50%. This result can be deduced from the distribution of dynamic loads. For 124.m88ksim, there is a steady increase in the number of dynamic loads available as the threshold decreases from 90% to 50%. There is a tapering off in speedup though, as more mispredictions are seen near a threshold of 50%. For 129.compress, the step in the distribution of dynamic loads from 80% to 70% is reflected in a corresponding step in speedup. Performance gains for 126.gcc are more reflective of the large number of dynamic load operations than of their predictability. Penalties for misprediction at the lower thresholds reduce speedup for 126.gcc. Benchmark 130.li, with a distribution of dynamic loads similar to 126.gcc, has lower performance due to fewer dynamic loads. Benchmark 134.perl clearly suffers from not having many dynamic loads. Benchmarks 099.go and 132.ijpeg do not have good predictability for load operations.

Based on these performance results, a predictability threshold of 70% appears to be a good selection. From the distribution of predictability for dynamic loads in Figure 2.9, a threshold 70% includes a large majority of the predictable dynamic loads.

Choosing a threshold of predictability lower than 70% results in a tapering off in performance for some benchmarks. This is due to both higher penalties for value misprediction and saturation of functional unit resources, resulting in fewer saved execution cycles.

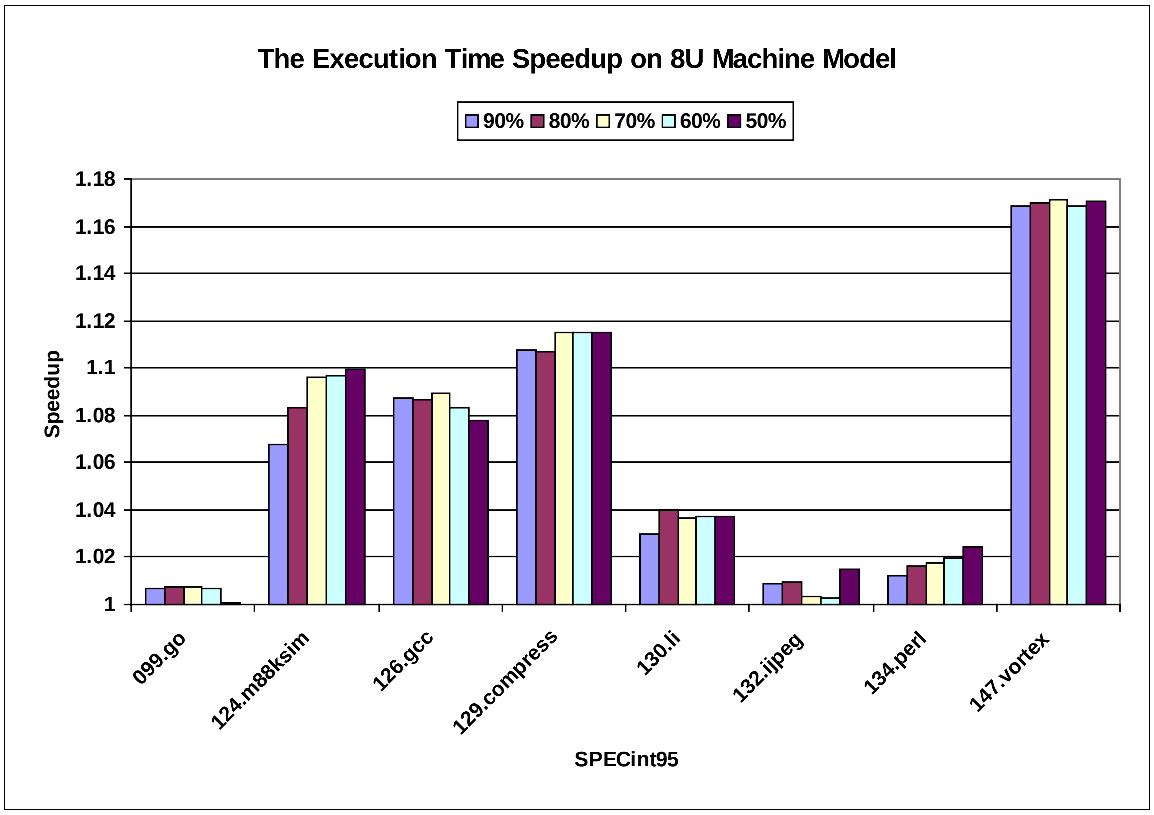


Figure 2.10 The execution time speedup for programs scheduled with VSS over without VSS. Prediction accuracy threshold values of 90%, 80%, 70%, 60% and 50% are used.

2.5 Summary

This chapter presents value speculation scheduling (VSS), a new technique for exploiting the value predictability of register-writing operations. This technique leverages advantages of both hardware schemes for value prediction and compiler schemes for exposing ILP. Dynamic value prediction is used to enable aggressive static schedules in which value dependent operations are speculated. In this way, VSS can be thought of as static ILP transformation that relies on dynamic value prediction hardware. The results for VSS presented in this chapter are impressive, especially when considering that only load operations are considered for value speculation. Chapter 3 will introduce a value speculation model to understand and improve the techniques for value speculation. By using the value speculation model, all true dependences among operations are considered for value prediction to obtain maximal benefits from value speculation.

Chapter 3

Modeling Value Speculation

Techniques for value speculation have been proposed for dynamically-scheduled machines [2], [3], [6], [8], [9], [22], [23], [26] and statically-scheduled machines [22], [23], [24], [25] to increase instruction-level parallelism by breaking flow (true) dependences and allowing value-dependent operations to be executed speculatively. Researchers have published many papers on designing value predictors yielding very high prediction accuracies [10], [11], [13], [15], [16]. Recently, the focus has shifted to the instruction selection techniques that choose important producer and consumer instructions for value prediction [8], [9], [24], [26]. The reason is that the effectiveness of value speculation relies not only on the predictability of operations, but also on the ability to shorten overall execution time, while encountering penalties for value misprediction. Several heuristics have been proposed to select operations for value prediction [8], [9], [24], [26], such as predicting operations at the top or in the middle of critical paths. However, it is unknown whether these heuristics do a good job of obtaining maximal benefits from value speculation.

To understand and improve the techniques for value speculation, we model value speculation as an optimal edge selection problem. Edges represent dependences between operations in a data dependence graph. The optimal edge selection problem involves finding an optimal (minimal) set of edges to break that achieves maximal benefits from value speculation, while taking the penalties for value misprediction into account. Based on three properties observed from the optimal edge selection problem, an efficient algorithm is designed using the techniques of branch-and-bound and memoization (a variation of dynamic programming) [21]. After running the optimal edge selection algorithm, several experimental results of modeling value speculation are presented in this chapter, including:

- The maximal benefits from value speculation on the 20 most heavily executed paths in the SPECint95 benchmarks.
- The impact of different penalties for branch misprediction on the benefits.
- The value prediction accuracy distribution and the location distribution of an optimal set of edges (dependences).
- The location distribution of the selected producer and consumer operations.
- The top five opcodes of the selected producer and consumer operations.

The remainder of this chapter is organized as follows. Section 3.1 briefly introduces different techniques for value speculation. Section 3.2 presents an optimal edge selection problem formally. Section 3.3 describes three properties observed from the optimal edge selection problem. Section 3.4 presents an optimal edge selection algorithm, with experimental results shown in Section 3.5. Section 3.6 concludes this chapter.

3.1 Introduction of Value Speculation

The techniques for value speculation in dynamically-scheduled and statically-scheduled machines are introduced as follows. In dynamically-scheduled machines [2], [3], there is an instruction window that maintains a pool of instructions waiting to be executed. All instructions in the instruction window dynamically form a data dependence graph. Without the value prediction technique, the dynamic scheduler selects an instruction to execute only if all of its operands are ready. However, by using value prediction to break flow dependences, the original data dependence graph can be collapsed and instructions can be speculatively executed even if their operands are not ready. Speculatively executed instructions must wait for verifying predicted values before their retirement. In the case of value misprediction, recovery mechanisms are required to re-execute instructions with correct operands. One recovery scheme utilizes the branch misprediction handling hardware [8] that is already in the dynamically-scheduled machine. All instructions following the incorrectly predicted instruction are re-fetched and re-executed. Another recovery mechanism is the selective re-issuing scheme [2], [3] to re-execute dependent instructions that are affected by incorrect predictions. The implementation of the selective re-issuing scheme is more complicated than that of the branch misprediction handling hardware.

For statically-scheduled machines, the compiler is responsible for forming a region of code and building the data dependence graph for all operations inside the region. The scheduler must honor all dependences among operations to generate a correct schedule. With the help of value prediction and value speculation, the scheduler can break true dependences and speculatively schedule value-dependent operations. The compiler inserts predicting operations, *LDPRED* [22], to load a prediction from the value predictor, and verifying operations, *BNE* (branch if not equal) [22], to compare the predicted value with the actual result. In the case of value misprediction, the compiler can provide recovery code [22] for re-executing operations, or advanced hardware can generate recovery code on the fly and execute recovery code on a separate compensation engine [25].

The challenge for value speculation is the combination of breaking true dependences among all dependences in the data dependence graph to reduce overall execution time, while also considering penalties resulting from value misprediction and the side effect of value-speculative execution. The penalties may include cycles for verifying value prediction, re-executing operations, I-Cache stalls due to re-fetching operations and aggressive speculative execution, D-Cache stalls due to more executed memory operations, and structural hazards [3] that come from the competition for machine resources (e.g., functional units, entries in the branch predictor) by non-speculative and speculative operations. Different recovery techniques for dynamically-scheduled and statically-scheduled machines have different penalties for value misprediction and value-speculative execution. The penalties under different recovery techniques for value speculation are compared in Table 3.1.

Table 3.1 Penalties under different recovery techniques for value speculation.

Recovery Techniques	**Dynamically-Scheduled Machines**		**Statically-Scheduled Machines**	
	Branch Misprediction Handling Hardware [8]	**Selective Re-issuing [2], [3]**	**Compiler-Generated Recovery Code [22]**	**Hardware-Generated Recovery Code [25]**
Penalties for Verifying Value Prediction	1 cycle (for comparing actual and predicted values) always + Flushing all pipeline stages when value is mispredicted	1 cycle (for comparing actual and predicted values) always	1 cycle (for comparing actual and predicted values) always + Flushing all pipeline stages when the BNE operation is mispredicted	1 cycle (for comparing actual and predicted values) always
Penalties for Re-execution	Re-executing all operations when value is mispredicted	Re-executing only affected operations when value is mispredicted	Re-executing only affected operations when value is mispredicted	Re-executing only affected operations when value is mispredicted (on a separate engine)
I-Cache Stalls	Re-fetching all operations when value is mispredicted + Side effect of speculative execution	Side effect of speculative execution	Fetching recovery code when value is mispredicted + Side effect of speculative execution	Side effect of speculative execution
D-Cache Stalls, Structure Hazards	Side effect of speculative execution	Side effect of speculative execution	Side effect of speculative execution	Side effect of speculative execution

As shown in Table 3.1, the scheme of compiler-generated recovery code for statically-scheduled machines has the penalties of one cycle for verifying the predicted value, the flushing cycles after the verifying operation (BNE) is mispredicted, the cycles for executing recovery code, additional I-Cache stalls for fetching recovery code, and extra stalls due to the impact of speculative execution on the I-Cache, D-Cache, and machine resources. In Table 3.1, some items of penalties are the same under different recovery mechanisms, but others are different.

3.2 An Optimal Edge Selection Problem

3.2.1 Terminology of Data Dependence Graphs

In related work [14], a dynamic data dependence graph is utilized to study the available parallelism with data value prediction. For modeling value speculation, the data dependence graph is heavily used as well. Terminology required on the data dependence graph is introduced as follows.

The data dependence graph that is formed in the instruction window or generated by an acyclic code scheduler is a directed acyclic graph (*DAG*). The data dependence graph is denoted by *DDG=(N, E)*, where N is the set of *Nodes* representing operations and E is the set of *Edges* representing dependences between operations. For an edge E_i, *Source*(E_i) is the source node of the edge E_i and *Sink*(E_i) is the sink node of the edge E_i. The types of edges include register flow (true) dependences, register anti- (false) dependences, register output (false) dependences, memory dependences, and control dependences [19]. Edges that are flow dependence types are candidates for the value speculation techniques to break. Each edge has a latency based on the dependence type.

Register flow and output dependences have latencies equal to the latencies of source operations. Register anti-, memory, and control dependences have latencies of zeros.

Each node has a *Height*, which is the latest scheduled cycle without delaying other operations. A top-down *depth-first-search (DFS)* algorithm [21] shown in Figure 3.1 can compute heights of all nodes in a data dependence graph. The running time of computing heights is O(|N| + |E|), where |N| is the number of nodes and |E| is the number of edges. The node also has a *Depth*, which is the earliest scheduled cycle of the operation. Depths are calculated by a bottom-up DFS algorithm, very similar to the algorithm shown in Figure 3.1. Only heights are used in this chapter. The maximal height of all nodes in the DDG represents the minimal cycles to execute or schedule all operations in the DDG. It is denoted by |*DDG*|, and called the *Length* or the *Height* of the DDG.

A *Critical Path* is the longest path from the starting nodes (of height 0) without predecessors to the ending nodes without successors in a data dependence graph. Based on the heights of nodes in the DDG, the critical path can be found using the algorithm shown in Figure 3.2. The algorithm of finding the critical path is similar to DFS, so its running time is O(|N| + |E|). The length of the critical path equals the length of the DDG (=|DDG|). Figure 3.3 shows examples of data dependence graphs with nodes that are laid out by heights and with edges that can be any dependence type. The critical path in the data dependence graph shown in Figure 3.3 consists of thick edges and thick-circled nodes.

```
// Compute heights of all nodes in a DDG

Compute_Height(DDG)
{
  // Step 1. Reset height of nodes and length of DDG.
  DDG->length = -1;
  For each node in DDG  {
    node->height = -1;
  }

  // Step 2. Compute height
  for each node in DDG  {
    height = Compute_Height(node);
    if(height > DDG->length)  {
      DDG->length = height;
    }
  }

  // Step 3. Reverse heights of all nodes, so heights are the
  //         latest scheduled cycle.
  for each node in DDG  {
    node->height = DDG->length – node->height;
  }
}

// Compute height of this node
int Compute_Height(node)
{
  // Step 1. If node has height, return its height.
  If (node->height != -1)  {
    return node->height;
  }

  // Step 2.  Get the max height from its successors.
  max_height = 0;
  for each succ_edge of node  {
    sink_node = succ_edge->sink;
    succ_height = Compute_Height(sink_node);
    new_height = succ_height + succ_edge->latency;
    if(new_height > max_height)  {
      max_height = new_height;
    )
  }

  if node has no succ_edge
    node->height = node->op->latency;
  else
    node->height = max_height;

  return node->height;
}
```

Figure 3.1 An algorithm of computing heights of all nodes in a data dependence graph.

```
// Find all critical paths in a DDG

Find_Critical_Path(DDG)
{
  // Step 1. Compute heights of nodes in the DDG.
  Compute_Height(DDG);

  // Step 2. Reset critical attributes of nodes and edges.
  for each node in DDG  {
    node->critical = false;

    for each succ_edge of node  {
      succ_edge->critical = false;
    }
  }

  // Step 3. Find critical paths from nodes with height 0.
  for each node in DDG  {
    if (node->height == 0)  {
      node->critical = true;
      Find_Critical_Path(node);
    }
  }
}

// Find critical path starting from node

Find_Critical_Path(node)
{
  for each succ_edge of node  {
    sink_node = succ_edge->sink;

    if (sink_node->height == (node->height + succ_edge->latency))  {
      succ_edge->critical = true;

      if(sink_node->critical == false)  {
        sink_node->critical = true;
        Find_Critical_Path(sink_node);
      }
    }
  }
}
```

Figure 3.2 An algorithm of finding critical paths in a data dependence graph.

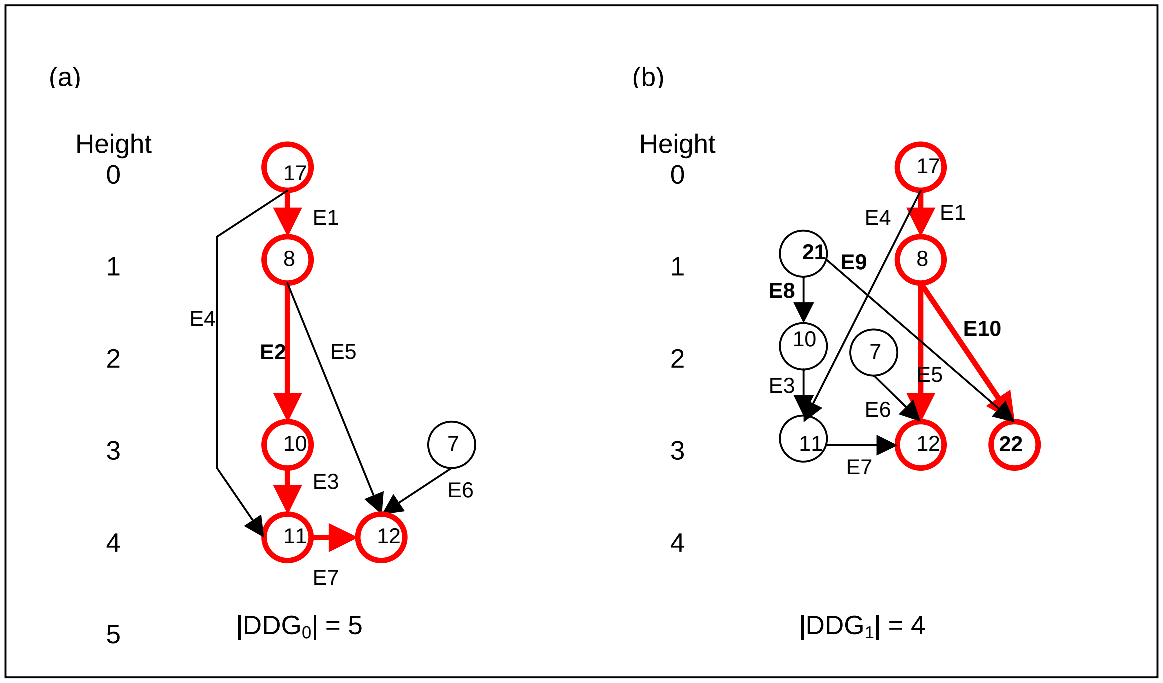


Figure 3.3 (a) A data dependence graph. (b) A modified data dependence graph after performing the value speculation transformation on E_2 (from node 8 to node 10). Thick edges and thick-circled nodes are on the critical path.

3.2.2 The Problem Statement

The model of value speculation is best illustrated by an example. For the data dependence graph shown in Figure 3.3(a), one edge E_2, from node 8 to node 10, is selected by the value speculation technique. In Figure 3.3(b), the *value speculation transformation* is performed, including breaking the edge E_2 (from node 8 to node 10), adding one predicting node 21 (LDPRED), adding one verifying node 22 (BNE), adding one edge E_8 (from node 21 to node 10), adding one edge E_9 (from node 21 to node 22), and adding one edge E_{10} (from node 8 to node 22). The predicting node loads a prediction from a value predictor, and feeds its result to node 10. The verifying node compares the predicted value from the predicting node and the actual result of node 8.

Note that the predicting and verifying nodes are explicit in statically-scheduled machines, but are implicit in dynamically-scheduled machines.

In Figure 3.3(a), the length of the DDG is 5 cycles. In Figure 3.3(b), after performing the value speculation transformation, the length of the *modified* DDG is reduced to 4 cycles. The modified DDG, denoted by DDG_n, is obtained after performing the value speculation transformation on *n* edges in the original DDG. The original DDG without performing the value speculation transformation on any edge is DDG_0.

In the case of value misprediction, penalties are incurred for recovery. The total penalty for mispredicting node N_i is denoted by *Penalty(N_i)*. It is assumed to be greater than zero. Based on Penalty(N_i), the penalty for mispredicting edge E_i, *Penalty(E_i),* is defined as follows:

$$\text{Penalty}(E_i) = \begin{cases} \text{Penalty(Source}(E_i)), \text{ if the source node of } E_i \text{ has not been predicted yet.} \\ 0, \text{ if the source node of } E_i \text{ has been predicted already.} \end{cases}$$

Because Penalty(N_i) is counted at most once in the proposed model, if the source node of E_i has not been predicted yet, Penalty(E_i) equals Penalty(Source(E_i)) after performing the value speculation transformation on the edge E_i. Otherwise, Penalty(E_i) is zero. Note that in the latter case of Penalty(E_i) equal to zero, the predicting node, the verifying node, and some new edges have been created already, so the value speculation transformation includes only breaking the selected edge and adding one edge from the predicting node to the sink node of the selected edge.

For an acyclic data dependence graph DDG=(N, E), find a minimal set of edges as $\{E_1, E_2, \ldots, E_{n-1}, E_n\}$, such that the benefit is maximal (and must be greater than zero) by performing the value speculation transformation on selected edges. The benefit for the DDG is defined as follows.

$$\begin{aligned}\text{Benefit(DDG)} &= \text{Benefit}(\text{DDG}_0)\\ &= \text{Execution\_cycles\_of\_DDG}_0 - \text{Execution\_cycles\_of\_DDG}_n\\ &= |\text{DDG}_0| - (|\text{DDG}_n| + \sum_{i=1}^{n} \text{Penalty}(E_i))\\ &= (|\text{DDG}_0| - |\text{DDG}_n|) - \sum_{i=1}^{n} \text{Penalty}(E_i)\\ &= \text{Cycle\_savings} – \text{Misprediction\_penalties}\end{aligned}$$

where

$\text{Penalty}(N_i) > 0$ for all nodes,

$$\text{Penalty}(E_i) = \begin{cases}\text{Penalty(Source(Ei)), if the source node of Ei has not been predicted yet.}\\ \text{0, if the source node of Ei has been predicted already.}\end{cases}$$

Figure 3.4 An optimal edge selection problem.

$$\begin{aligned}\text{Penalty}(N_i) = {} & \text{Value\_misprediction\_rate * Cycles\_of\_recovery\_code +}\\ & \text{BNE\_branch\_misprediction\_rate * Stall\_cycles\_of\_mispredicted\_branch}\end{aligned}$$

where

$\text{Cycles\_of\_recovery\_code} = |\text{DDG}_0| – \text{Height}(N_i)$,

Stall_cycles_of_mispredicted_branch = 2, 5, or 10,

Value misprediction rates and BNE branch misprediction rates come from profile results.

Figure 3.5 Penalties of nodes.

Using the introduced terminology, value speculation is modeled as an optimal edge selection problem that is formally presented in Figure 3.4. The optimal edge selection problem asks for finding a minimal set of edges such that the benefit is maximal (and must be greater than 0) by performing the value speculation transformation on selected edges.

Some assumptions and limitations of the proposed value speculation model are as follows:

- The data dependence graph must be a directed acyclic graph (DAG). The DDG is constructed for operations in the instruction window of dynamically-scheduled machines, or for operations in a linear path (trace) of basic blocks in a program for statically-scheduled machines.
- The selected edge must belong to the original set of edges, and must be a flow (true) dependence type.
- In the optimal edge selection problem, the latencies of edges and the penalties for mispredicting nodes must be known beforehand and fixed all the time. In this chapter, the value speculation technique on a statically-scheduled machine with compiler-generated recovery code is experimented. According to Table 3.1, the penalties for mispredicting nodes are modeled by the equation in Figure 3.5. (The penalties in other recovery schemes shown in Table 3.1 can be modeled as well.) Note that the one-cycle penalty for comparing the predicted value with the actual result does not appear in the equation, because the verifying node has already been inserted in the data dependence graph. The penalties of the I-Cache stalls, D-Cache

stalls, and structural hazards are ignored. In the equation, the value misprediction rates and the BNE branch misprediction rates come from profile results.

- Machine resources are not taken into account in the optimal edge selection problem. Unlimited resources are assumed to be available for the value speculation techniques.
- In dynamically-scheduled machines, instructions shift into and out of the instruction window every cycle. However, the proposed value speculation model focuses only on a static data dependence graph that is composed of the instructions in the current instruction window.

3.3 Three Properties Observed from the Optimal Edge Selection Problem

The optimal edge selection problem presented in Figure 3.4 can be solved by a brute-force method that measures the benefits of all possible edge selections. For |E| edges, the brute force method must try $2^{|E|}$ combinations. However, from observing the optimal edge selection problem, there exist some properties for us to design an efficient algorithm.

The first observation is that because the process of the value speculation transformation is deterministic, the final DDG_n should be the same regardless of the order of the value speculation transformation performed on the selected edges in the DDG_0. Therefore, the computation of the benefit on all selected edges can be decomposed into calculating the benefit difference of each selected edge. Property 1 is shown in Figure 3.6, and its proof appears as follows.

Property 1: Decomposition

Let Benefit_Difference(E_i) = $|DDG_{i-1}|$ - $|DDG_i|$ - Penalty(E_i). Then, for a set of edges $\{E_1, E_2, \ldots, E_{n-1}, E_n\}$, the benefit for the DDG_0 is the summation of all benefit differences.

Figure 3.6 Property 1 of the optimal edge selection problem: decomposition.

Proof of Property 1:

For the presentation, the index of the edges $\{E_1, E_2, \ldots, E_{n-1}, E_n\}$ is coincidently the same as the order when they are selected. The benefit for the DDG_0 after performing the value speculation transformation on $\{E_1, E_2, \ldots, E_{n-1}, E_n\}$ is denoted by

Benefit(DDG_0)

$$= (|DDG_0| - |DDG_n|) - \sum_{i=1}^{n} \text{Penalty}(E_i)$$

$$= (|DDG_0| - |DDG_1|) + (|DDG_1| - |DDG_2|) + \ldots + (|DDG_{n-1}| - |DDG_n|) - \sum_{i=1}^{n} \text{Penalty}(E_i)$$

$$= (|DDG_0| - |DDG_1| - \text{Penalty}(E_1)) + (|DDG_1| - |DDG_2| - \text{Penalty}(E_2)) + \ldots + (|DDG_{n-1}| - |DDG_n| - \text{Penalty}(E_n))$$

$$= \sum_{i=1}^{n} (|DDG_{i-1}| - |DDG_i| - \text{Penalty}(E_i))$$

$$= \sum_{i=1}^{n} \text{Benefit\_Difference}(E_i). \qquad \#$$

The second observation is the optimal substructure [21] of the optimal edge selection problem. For the data dependence graph shown in Figure 3.3(a), if the set of $\{E_2, E_5\}$ is an optimal solution for the DDG_0, after performing the value speculation transformation on {E_2}, {E_5} will be an optimal solution for the DDG_1 shown in Figure 3.3(b). (Note that for the modified DDG, we restrict that the candidate edge must still belong to the original set of edges in the DDG_0, and must be a true dependence type.) Property 2 is shown in Figure 3.7, and its proof appears as follows.

Property 2: Optimal Substructure

For an optimal set of edges {E_1, E_2, …, E_{n-1}, E_n} for the DDG_0, after performing the value speculation transformation on a subset of optimal edges, the remaining edges in the optimal set of edges is an optimal solution for the modified DDG. So, the problem of each modified DDG is also an optimal edge selection problem.

Figure 3.7 Property 2 of the optimal edge selection problem: optimal substructure.

Proof of Property 2: (By Contradiction)

Because {E_1, E_2, …, E_{n-1}, E_n } is an optimal solution for the DDG_0, {E_1, E_2, …, E_{n-1}, E_n } should be the minimal set of edges that yield the highest positive benefit for the DDG_0. Without loss of generality, {E_1, E_2, …, E_{n-1}, E_n} is split into two sets of edges, {E_1, E_2, …, E_k} and {E_{k+1}, …, E_{n-1}, E_n}. For the presentation, the index of the edges {E_1, E_2, …, E_n} is coincidently the same as the order when they are selected. From Property 1, the maximal benefit for the DDG_0 is denoted by $Benefit_{old}(DDG_0)$

$$= \sum_{i=1}^{n} (|DDG_{i-1}| - |DDG_i| - Penalty(E_i))$$

$$= \sum_{i=1}^{k} (|DDG_{i-1}| - |DDG_i| - Penalty(E_i)) + \sum_{i=k+1}^{n} (|DDG_{i-1}| - |DDG_i| - Penalty(E_i)).$$

Let $\text{Benefit}_{old}(\text{DDG}_k) = \sum_{i=k+1}^{n} (|\text{DDG}_{i-1}| - |\text{DDG}_i| - \text{Penalty}(\text{E}_i))$. Then,

$\text{Benefit}_{old}(\text{DDG}_0) = \sum_{i=1}^{k} (|\text{DDG}_{i-1}| - |\text{DDG}_i| - \text{Penalty}(\text{E}_i)) + \text{Benefit}_{old}(\text{DDG}_k)$.

Property 2 states that $\{E_{k+1}, \ldots, E_{n-1}, E_n\}$ must be an optimal solution for the DDG_k. We will prove it by the following four cases.

Case 1. If we assume that $\{E_{k+1}, \ldots, E_{n-1}, E_n\}$ does not yield the highest benefit for the DDG_k, one new $\text{Benefit}_{new}(\text{DDG}_k)$ can be found to be higher than the $\text{Benefit}_{old}(\text{DDG}_k)$. Adding $\sum_{i=1}^{k} (|\text{DDG}_{i-1}| - |\text{DDG}_i| - \text{Penalty}(\text{E}_i))$ and the $\text{Benefit}_{new}(\text{DDG}_k)$ together, one new $\text{Benefit}_{new}(\text{DDG}_0)$ can be found to be higher than the $\text{Benefit}_{old}(\text{DDG}_0)$. This contradicts that the $\text{Benefit}_{old}(\text{DDG}_0)$ should be maximal. Therefore, we cannot find other set of edges for the DDG_k to have higher benefits than $\sum_{i=k+1}^{n} (|\text{DDG}_{i-1}| - |\text{DDG}_i| - \text{Penalty}(\text{E}_i))$.

Case 2. If we assume that $\{E_{k+1}, \ldots, E_{n-1}, E_n\}$ for the DDG_k is not the minimal set of edges that yield the maximal benefit (= $\text{Benefit}_{old}(\text{DDG}_k)$), a smaller set of edges $\{E_{k'+1}, \ldots, E_{n'-1}, E_{n'}\}$ can be found to have the same benefit (=$\text{Benefit}_{old}(\text{DDG}_k)$). Combining $\{E_1, E_2, \ldots, E_k\}$ and $\{E_{k'+1}, \ldots, E_{n'-1}, E_{n'}\}$ forms a smaller set of edges for the DDG_0 that yield the benefit equal to the $\text{Benefit}_{old}(\text{DDG}_0)$. This contradicts that $\{E_1, E_2, \ldots, E_{n-1}, E_n\}$ should be the minimal set of edges for the DDG_0. Therefore, $\{E_{k+1}, \ldots, E_{n-1}, E_n\}$ is the minimal set of edges that yield the maximal benefit (= $\text{Benefit}_{old}(\text{DDG}_k)$) for the DDG_k.

Case 3. If we assume that $\sum_{i=k+1}^{n}$ ($|DDG_{i-1}|$ - $|DDG_i|$ - Penalty(E_i)) is zero, performing value speculation transformation on {E_1, E_2, …, E_k} for the DDG_0 will obtain the benefits

$= \sum_{i=1}^{k}$ ($|DDG_{i-1}|$ - $|DDG_i|$ - Penalty(E_i))

$= \sum_{i=1}^{k}$ ($|DDG_{i-1}|$ - $|DDG_i|$ - Penalty(E_i)) + 0

$= \sum_{i=1}^{k}$ ($|DDG_{i-1}|$ - $|DDG_i|$ - Penalty(E_i)) + $\sum_{i=k+1}^{n}$ ($|DDG_{i-1}|$ - $|DDG_i|$ - Penalty(E_i))

$= \sum_{i=1}^{n}$ ($|DDG_{i-1}|$ - $|DDG_i|$ - Penalty(E_i))

= $\text{Benefit}_{old}(DDG_0)$.

{E_1, E_2, …, E_k} and {E_1, E_2, …, E_{n-1}, E_n} yield the same benefits. However, {E_1, E_2, …, E_{n-1}, E_n} contains more edges than {E_1, E_2, …, E_k}. This contradicts that {E_1, E_2, …, E_{n-1}, E_n} should be the minimal set of edges for the DDG_0. Therefore, $\sum_{i=k+1}^{n}$ ($|DDG_{i-1}|$ - $|DDG_i|$ - Penalty(E_i)) is not zero.

Case 4. If we assume that $\sum_{i=k+1}^{n}$ ($|DDG_{i-1}|$ - $|DDG_i|$ - Penalty(E_i)) is negative, performing value speculation transformation on {E_1, E_2, …, E_k} for the DDG_0 will obtain the benefits

$= \sum_{i=1}^{k}$ ($|DDG_{i-1}|$ - $|DDG_i|$ - Penalty(E_i))

$> \sum_{i=1}^{k}$ ($|DDG_{i-1}|$ - $|DDG_i|$ - Penalty(E_i)) + $\sum_{i=k+1}^{n}$ ($|DDG_{i-1}|$ - $|DDG_i|$ - Penalty(E_i))

$= \sum_{i=1}^{n}$ ($|DDG_{i-1}|$ - $|DDG_i|$ - Penalty(E_i)) = $\text{Benefit}_{old}(DDG_0)$.

$\{E_1, E_2, \ldots, E_k\}$ yields a higher benefit for the DDG_0 than $\{E_1, E_2, \ldots, E_{n-1}, E_n\}$ does. This contradicts that $\{E_1, E_2, \ldots, E_{n-1}, E_n\}$ should yield the highest benefit for the DDG_0. Therefore, $\sum_{i=k+1}^{n}$ ($|DDG_{i-1}|$ - $|DDG_i|$ - Penalty(E_i)) is not negative.

Summary: From the abovementioned four cases, we know that for the DDG_k, $\{E_{k+1}, \ldots, E_{n-1}, E_n\}$ is the minimal set of edges (from Case 2) that yield the highest (from Case 1) and positive (from Case 3 and Case 4) benefit. Therefore, $\{E_{k+1}, \ldots, E_{n-1}, E_n\}$ is an optimal solution for the DDG_k. Also, any subset of optimal edges for the DDG_0 is an optimal solution for the corresponding modified DDG. So, the problem of each modified DDG is also an optimal edge selection problem. #

The third observation is that if an optimal solution (an optimal set of edges) exists for the DDG_0, there must be at least one edge from the optimal set of edges on the critical path of the DDG_0; otherwise the length of the critical path in the DDG_0 will never be shortened, and the overall benefit will never be greater than 0. From Property 2, the problem of each modified DDG is also an optimal edge selection problem. So, based on the same reason, there must be at least one optimal edge on the critical path in each modified DDG. Property 3 is stated as follows and its proof stands from our discussion.

Property 3: Critical Edge Selection

For an optimal set of edges for the DDG_0, there exists a sequence of edges, such that each selected edge is on the critical path of each modified DDG.

Figure 3.8 Property 3 of the optimal edge selection problem: critical edge selection.

3.4 An Optimal Edge Selection Algorithm

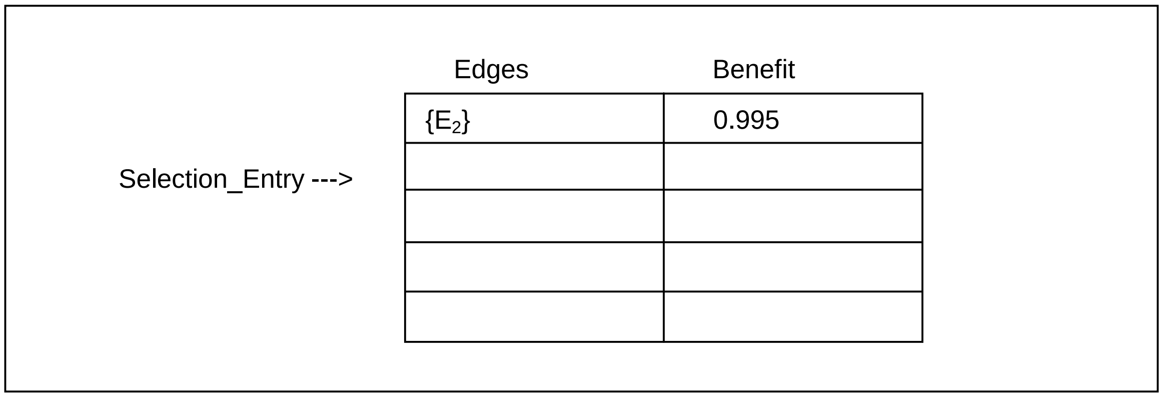
3.4.1 The Algorithm

Based on the three properties introduced in Section 3.3, an optimal edge selection algorithm shown in Figure 3.9 is designed using the techniques of branch-and-bound and memoization (a variation of dynamic programming) [21]. The algorithm employs top-down recursion to try different sets of edges. A selection table shown in Figure 3.10 records each edge selection and its corresponding benefit. In Figure 3.10, the first selection entry contains an edge {E_2} and its benefit is 0.995 cycles obtained by performing the value speculation transformation on E_2 in the data dependence graph shown in Figure 3.3(a). The optimal edge selection algorithm needs to search the table to know if the same edge selection has been tried already. Thus, the selection table is managed as a hashed table to reduce the table lookup time. Hashing every id of selected edges generates the table index. To reduce the memory space for storing the selection entries, bit vectors can be used to record edges.

In Figure 3.9, the *Optimal_Edge_Selection_Algorithm* initializes the current maximal benefit, *current_max_benefit*, to 0.0 and sets the best selection entry, *best_se*, to NULL. The *current_max_benefit* indicates the maximal benefit that can be obtained for the data dependence graph, *ddg*. The *best_se* maintains a linked list of edge selections that yield the maximal benefits. After finding the critical paths in the *ddg*, the algorithm calls the *Selection_Algorithm* with parameters of the *ddg* and the *Selection_Entry *se* that points to the previous selection entry. The *Selection_Algorithm* contains a one-level loop to iterate each candidate edge in the current *ddg*.

```
double current_max_benefit;
Selection_Entry *best_se;

Optimal_Edge_Selection_Algorithm(Data_Dependence_Graph *ddg)
{
   current_max_benefit = 0.0;
   best_se = NULL;
   Find_Critical_Path(ddg);
   Selection_Algorithm(ddg,NULL);
}
Selection_Algorithm(Data_Dependence_Graph *ddg, Selection_Entry *se)
{
   For each candidate edge in ddg
   {
      // Step 1. Filter candidate edges
      If edge is not on critical path of ddg, continue.
      If edge->sink_node->height is less than 2 or greater than
            (ddg->length)-2, continue.

      // Step 2. Generate a new edge selection
      new_edges = edge ∪ se->edges;
      If the selection table has an entry of new_edges already,
            continue.
      Get a new Selection_Entry *new_se from the selection table.
      new_se->edges = new_edges;

      // Step 3. Calculate benefit difference
      old_length = ddg->length;
      old_benefit = se->benefit;
      Perform value speculation transformation on edge in ddg.
      Find_Critical_Path(ddg);
      new_length = ddg->length;
      benefit_difference = (old_length - new_length) - penalty(edge);
      new_se->benefit = old_benefit + benefit_difference;
      Update current_max_benefit and best_se.

      // Step 4. Recursive call
      possible_max_benefit = new_se->benefit + possible_benefit(ddg);
      If (possible_max_benefit >= current_max_benefit) {
         Selection_Algorithm(ddg,new_se);
      }


       // Step 5. Undo changes of ddg made in this level
      Undo changes of ddg made in this level.
  }
}
```

Figure 3.9 An optimal edge selection algorithm.


Selection_Entry --->

Edges	Benefit
$\{E_2\}$	0.995

Figure 3.10 A selection table. Each selection entry records a set of edges and its corresponding benefit.

In Step 1, the algorithm filters out candidate edges based on two criteria. First, from Property 3 of critical edge selection, the algorithm can try only the candidate *edge* on the critical path to find an optimal solution. Second, if the height of the sink node of the *edge* is less than 2 or greater than |*ddg*|-2, the *edge* is too shallow or too deep to gain benefits from value speculation, so the *edge* can be skipped. (The length of the modified data dependence graph cannot be reduced after performing the value speculation transformation on shallow or deep edges, because the predicting and verifying nodes and new edges are added.)

In Step 2, combining the previous edge selection, *se->edges*, and the candidate *edge* forms a new edge selection, *new_edges*. The selection table is searched to check if a selection entry with the same *new_edges* already exists. If so, the algorithm continues to the next loop iteration. Otherwise, the table gets an empty selection entry, *new_se*, and stores the new edge selection, *new_edges*, to that entry.

In Step 3, the current *ddg* is updated by performing the value speculation transformation on the *edge*, including breaking the candidate *edge* and adding necessary nodes and edges. Then, from Property 1 of decomposition, the new benefit is calculated as the sum of the *old_benefit* (= *se->benefit*) and the *benefit_difference*, and is saved to the entry, *new_se->benefit*. If the new benefit is greater than the *current_max_benefit*, the *current_max_benefit* is updated and the *best_se* points to the *new_se*. If the benefits are the same, the *new_se* is attached to the linked list by searching from the *best_se*.

In Step 4, the possible maximal benefit, *possible_max_benefit*, is calculated as the new benefit plus the possible benefit for the current *ddg* by removing all remaining candidate edges temporarily. The recursion continues if the *possible_max_benefit* can beat or equal the *current_max_benefit*.

In Step 5, all changes of the *ddg* made in this level are undone, such that the *Selection_Algorithm* is ready to test the next candidate edge. After the *Selection_Algorithm* stops, if the *current_max_benefit* is greater than 0, an optimal solution is found by searching the linked list from the *best_se* to get a minimal set of edges that yield the maximal benefit (=*current_max_benefit*).

3.4.2 Running Time Analysis

To analyze the complexity of the proposed optimal edge selection algorithm, a call graph of the *Selection_Algorithm* for one data dependence graph in 129.compress is shown in Figure 3.11. Each node in the call graph represents one instance of the *Selection_Algorithm*. Inside each node, the first number is the called order of the *Selection_Algorithm*, and the second number is the corresponding benefit that the *Selection_Algorithm* finds after performing value speculation transformation on one specific edge selection. Having 12 candidate edges in the data dependence graph for the call graph shown in Figure 3.11, the *Selection_Algorithm* is called only 29 times. Note that a brute-force method needs to try $2^{12} = 4096$ combinations to find an optimal solution for 12 candidate edges. This shows that the *Selection_Algorithm* is quite efficient to find an optimal solution that yields the maximal benefit. In the call graph shown in Figure 3.11, the function call in the first level corresponds to the DDG_0, the function calls in the second level correspond to the DDG_1, and so on. Some function calls make recursions, and some function calls stop in certain levels of the modified DDG. This is due to the branch-and-bound condition in Step 4 of Figure 3.9. If the possible maximal benefit is less than the current maximal benefit, the recursion stops.

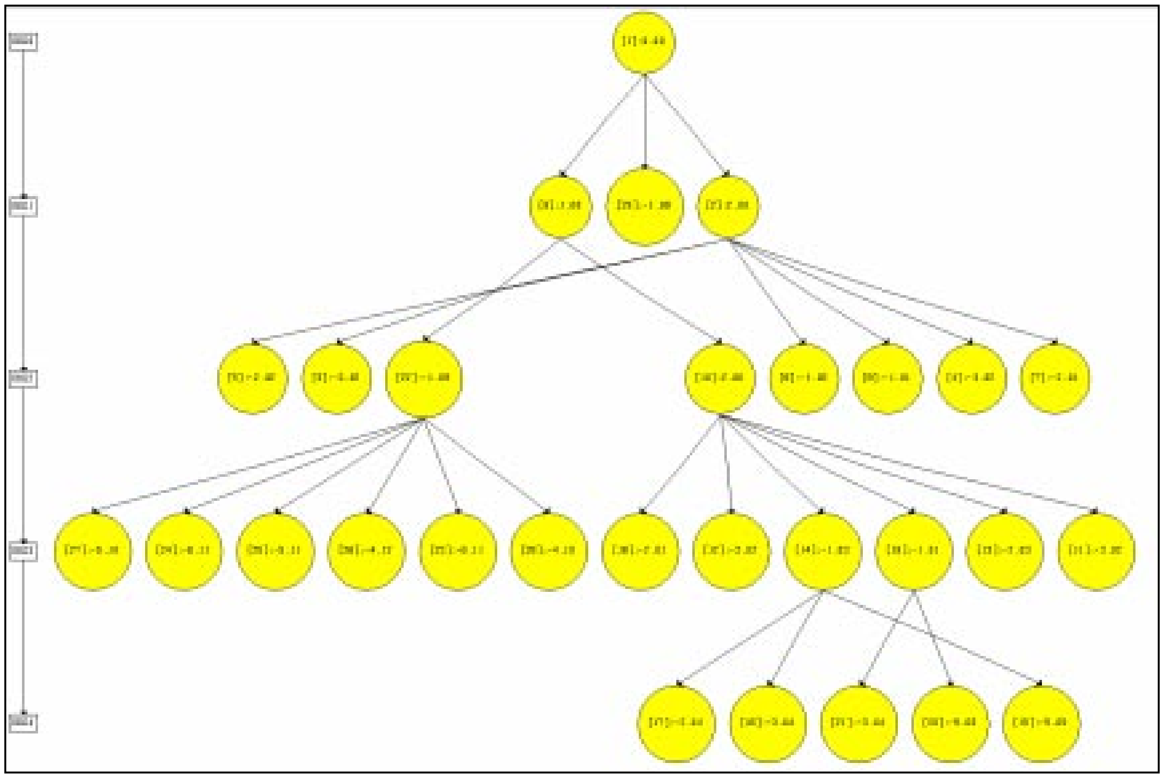


Figure 3.11 A call graph of the *Selection_Algorithm* for one data dependence graph in 129.compress. In each node, the first number is the called order, and the second number is the corresponding benefit that the *Selection_Algorithm* finds.

The running time of the proposed optimal edge selection algorithm is proportional to the number of selection entries that it tries multiplied by the running time of finding critical paths, which is O(|N| + |E|), in the *Selection_Algorithm*. In the best case where penalties of nodes are very large, the *Selection_Algorithm* does not continue the recursion after trying one edge in the *Selection_Algorithm*. Also, shallow, deep, or non-critical edges are skipped in Step1, so in the best case the *Selection_Algorithm* tries less than |E| combinations and then stops. The best-case running time is O(|E|) * O(|N| + |E|). In the worst case where penalties of nodes are very small, the algorithm always continues the recursion and tries almost all combinations of edges. Thus, the worst-case running time is exponential, with $O(2^{|E|})$ * O(|N| + |E|). However, because not all operations are highly predictable and the penalties of nodes vary, the average running time of the proposed algorithm is observed to be polynomial and efficient in our experiments.

Figure 3.12 shows the empirical running time analysis of the optimal edge selection algorithm applied to the data dependence graphs of the 20 most heavily executed paths selected from each SPECint95 benchmark. In Figure 3.12, the x-axis is the number of candidate edges, and the y-axis is the number of combinations that the *Selection_Algorithm* tries. From Figure 3.12, most benchmarks have less than 40 candidate edges for the *Selection_Algorithm* to try, except that 132.ijpeg has up to 160 candidate edges. With so many candidate edges, the algorithm tries less than 10,000 combinations for all data dependence graphs except one graph in 124.m88ksim that yields 14,599 combinations. From the data points in Figure 3.12, the average trend line is $y = 0.0016x^3 - 0.67x^2 + 67.349x - 306.35$, so the empirical average running time is $O(|E|^3)$ * O(|N| + |E|). Also, the empirical worst-case trend line is $y = 0.1012x^4 - 5.1062x^3 +$

$80.286x^2 - 416.35x + 467.08$. In Chapter 4, the value prediction accuracy heuristics are proposed to speed up the optimal edge selection algorithm by more than 80% and still find good benefits from value speculation. To implement the optimal edge selection algorithm in a production compiler, a bail-out mechanism can be employed in the *Selection_Algorithm* based on an upper bound of combinations that the algorithm can try, so the running time is controlled within a certain limit.

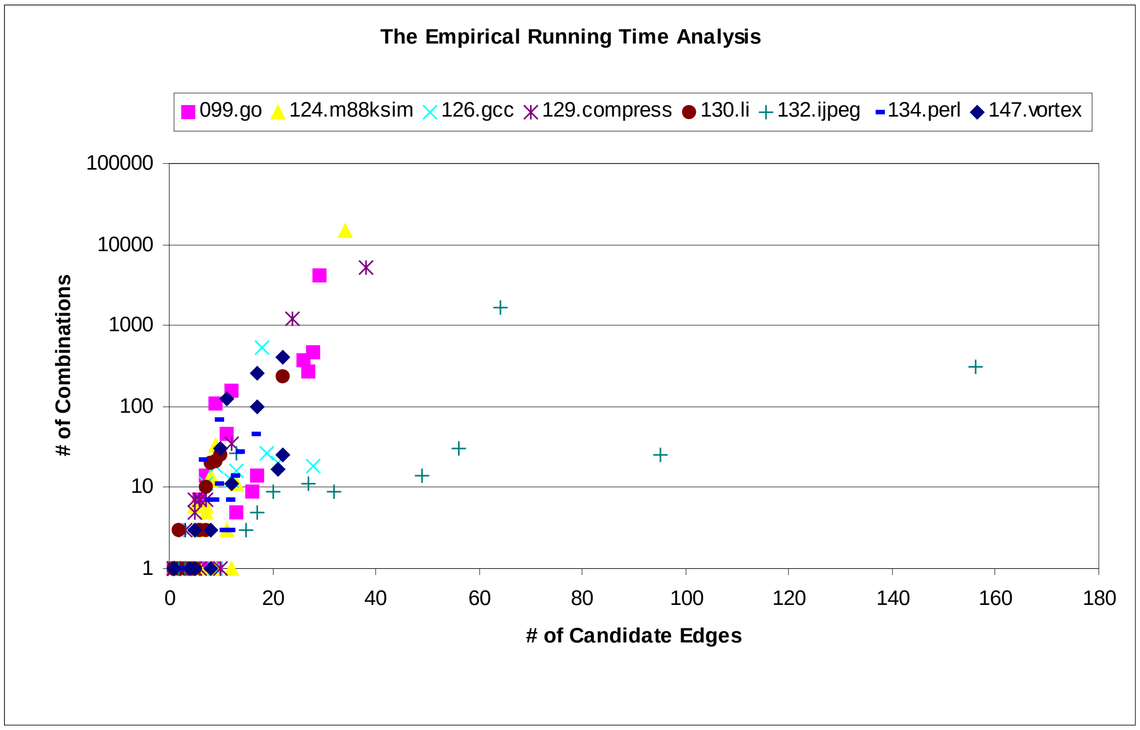


Figure 3.12 The empirical running time analysis of the optimal edge selection algorithm. For all data points, the average-case complexity is $y = 0.0016x^3 - 0.67x^2 + 67.349x - 306.35$. The worst-case complexity is $y = 0.1012x^4 - 5.1062x^3 + 80.286x^2 - 416.35x + 467.08$.

3.5 Experimental Results

In this section, the optimal edge selection algorithm was experimented for the value speculation scheme on a statically-scheduled machine with compiler-generated recovery code [22]. The target architecture is a VLIW machine model based on the Hewlett-Packard Laboratories HPL-PD architecture [20]. All operations have a one-cycle latency except for load (two cycles), floating-point add (two cycles), floating-point subtract (two cycles), floating-point multiply (threes cycles), and floating-point divide (three cycles). All SPECint95 programs were compiled with classic optimizations by the IMPACT compiler from the University of Illinois [18] and converted to the Rebel textual intermediate representation by the Elcor compiler from Hewlett-Packard Laboratories [19]. Then, the LEGO compiler, developed at North Carolina State University, was used to insert profiling code, form treegions, and schedule operations [17]. After instrumentation for profiling, intermediate code from the LEGO compiler was converted to C code. Executing the resultant C code generated profiling data.

Several experimental results are presented in this section. Section 3.5.1 presents the results of value profiling and branch profiling, which are used to model penalties for mispredicting nodes. Section 3.5.2 shows the maximal benefit from value speculation on the 20 most heavily executed paths in the SPECint95 benchmarks. Section 3.5.3 analyzes the selected edges, producer operations, and consumer operations.

3.5.1 Results of Value Profiling and Branch Profiling

As discussed in Section 3.2, to model the penalties for mispredicting nodes, the value misprediction rates and the BNE branch misprediction rates are required and provided by program profiling [4], [7] in the experiments.

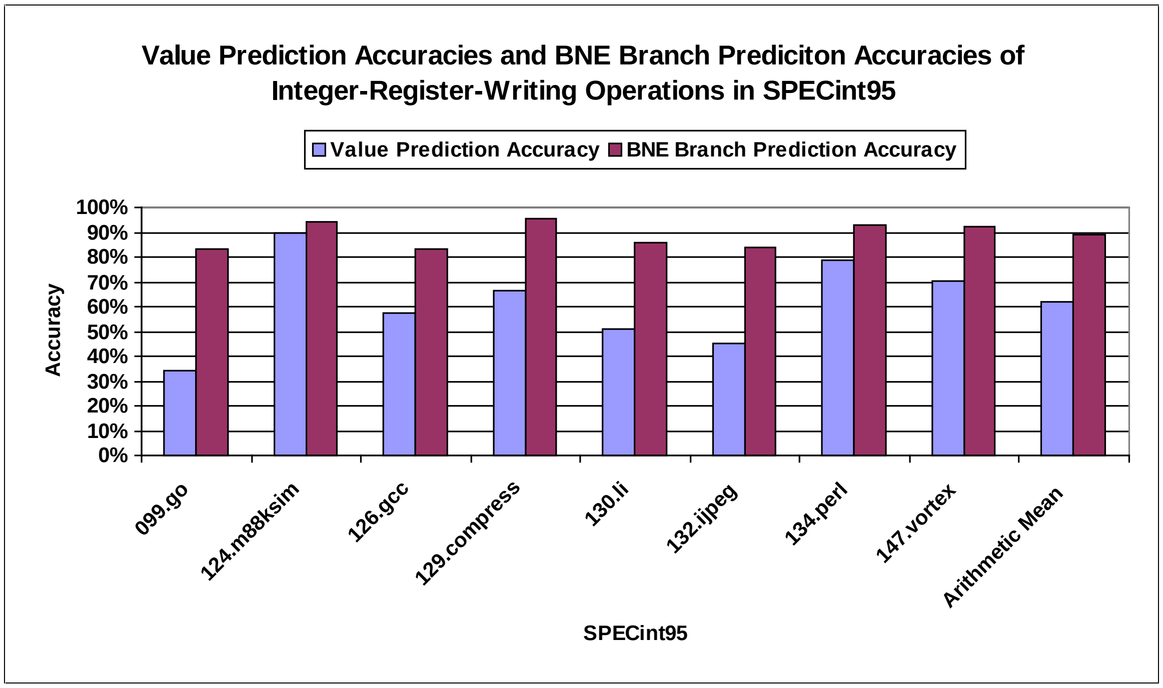


Figure 3.13 Value prediction accuracies and BNE branch prediction accuracies of integer-register-writing operations in the SPECint95 benchmarks.

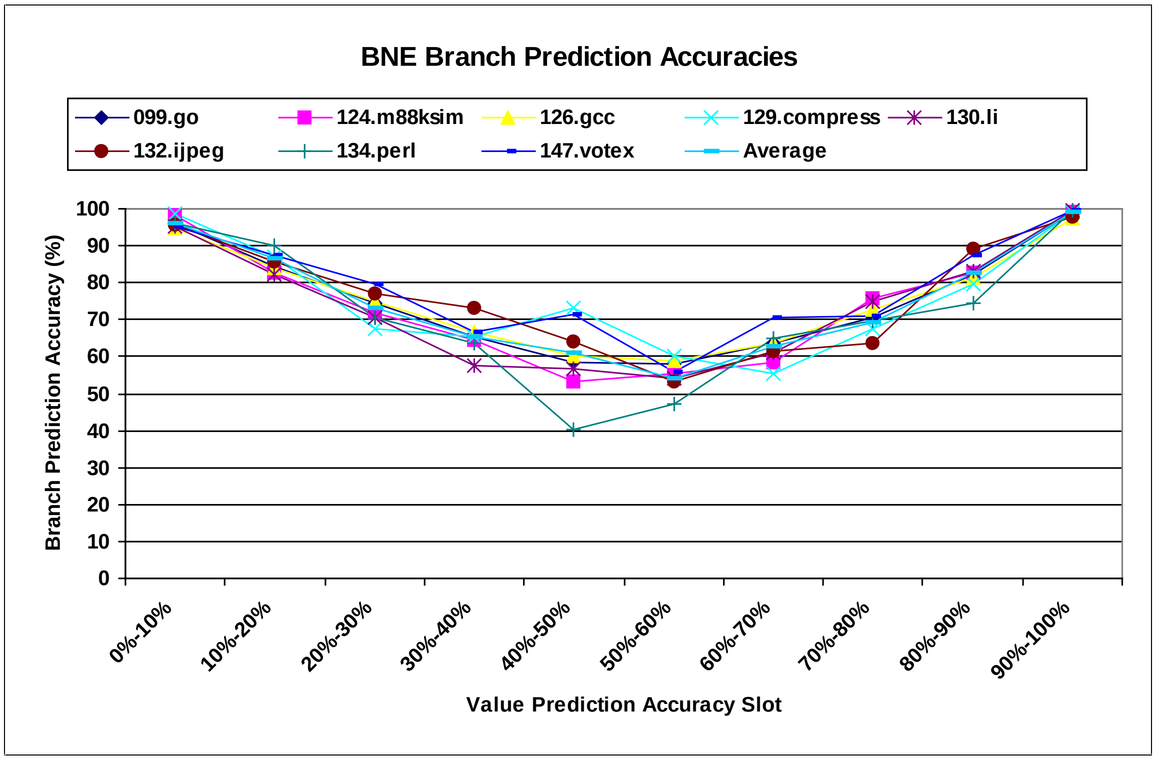


Figure 3.14 BNE branch prediction accuracies sorted by their corresponding value prediction accuracies.

A hybrid value predictor [13], [22], which contains stride [4], [10], [13] and context-based value predictors [10], [11], was used to profile all integer-register-writing operations. The value prediction table size equals the number of all integer-register-writing operations in each SPECint95 program. For context-based value predictors, the entry in the first level table records one actual result that indexes a local second level table with 16 entries. Simple two-bit counters were used to predict the BNE branches for verifying value prediction. Figure 3.13 shows the value prediction accuracies and the BNE branch prediction accuracies for integer-register-writing operations in the SPECint95 benchmarks. The average value prediction accuracy is 61.62%, lower than the average BNE branch prediction accuracy of 88.81%. In general, the BNE branches have high predictability, so that the processor can accurately predict to execute recovery code. The correct BNE branch prediction avoids the penalties for flushing pipeline stages in the case of branch misprediction.

The average BNE branch prediction accuracies are presented in Figure 3.14 by sorting them based on the corresponding value prediction accuracies. For highly predictable operations and highly unpredictable operations, the BNE branches have very high predictability. For operations with medium value prediction accuracies around 50%, their BNE branch prediction accuracies are the lowest. In 134.perl, the BNE branches that correspond to the value prediction accuracy between 40% and 60% have the branch prediction accuracy lower than 50%. Predicting operations with low BNE branch predictability may lead to high penalties for value misprediction. In Figure 3.14, the data points that correspond to value prediction accuracies lower than 10% or higher than 90% are very close to each other. However, the data points that correspond to value prediction accuracies between 30% and 60% are very distant. The reason is that operations tend to be highly predictable or highly unpredictable, so there are many operations that contribute to the average accuracy and the variance is small among SPECint95 benchmarks. Operations with medium value predictability are very few, so the variation of the average branch prediction accuracy among different benchmarks is large.

3.5.2 Maximal Benefits from Value Speculation

For further experiments of the optimal edge selection algorithm, the 20 most heavily executed paths were chosen from each SPECint95 benchmark. Penalties for mispredicting nodes were modeled based on the profile results and three different branch misprediction penalties of 2, 5, and 10 cycles. Running the optimal edge selection algorithm generates the results of an optimal set of edges and the corresponding maximal benefit. Analyzing the results helps us realize the potential of value speculation. Figure 3.15 shows the number of improved paths under different branch misprediction penalties. A path is counted as an improved path if the optimal edge selection algorithm can find a positive maximal benefit. Using branch misprediction penalties of 2, 5, and 10 cycles, the average numbers of improved paths are 5.875, 5.375, and 5.25 out of 20 paths. The larger branch misprediction penalties, the fewer paths can be improved by value speculation. In the SPECint95 benchmarks, 124.m88ksim has the most improved paths of 11. However, 132.ijpeg and 099.go have one or no paths improved by value speculation. The other benchmarks have around six improved paths.

Figure 3.16 shows the speedup of value speculation, which is measured as the maximal benefit divided by the minimal cycles of executing the original data dependence graph. As the branch misprediction penalty increases, the speedup decreases. When using a 10-cycle branch misprediction penalty, 124.m88ksim gets the highest speedup of 25.57%. 147.vortex has the second highest speedup of 16%. 126.gcc, 129.compress, 130.li, and 134.perl have significant speedups around 9% that are available for the value speculation techniques to exploit. The average speedup is 9.61% for all SPECint95 benchmarks. 099.go and 132.ijpeg with no or small speedups are not good candidates for value speculation. The results of Figures 10 and 11 suggest that if the penalties for branch misprediction are high, the optimization opportunities for the value speculation techniques will decrease, and the value speculation technique needs to carefully select and break dependences.

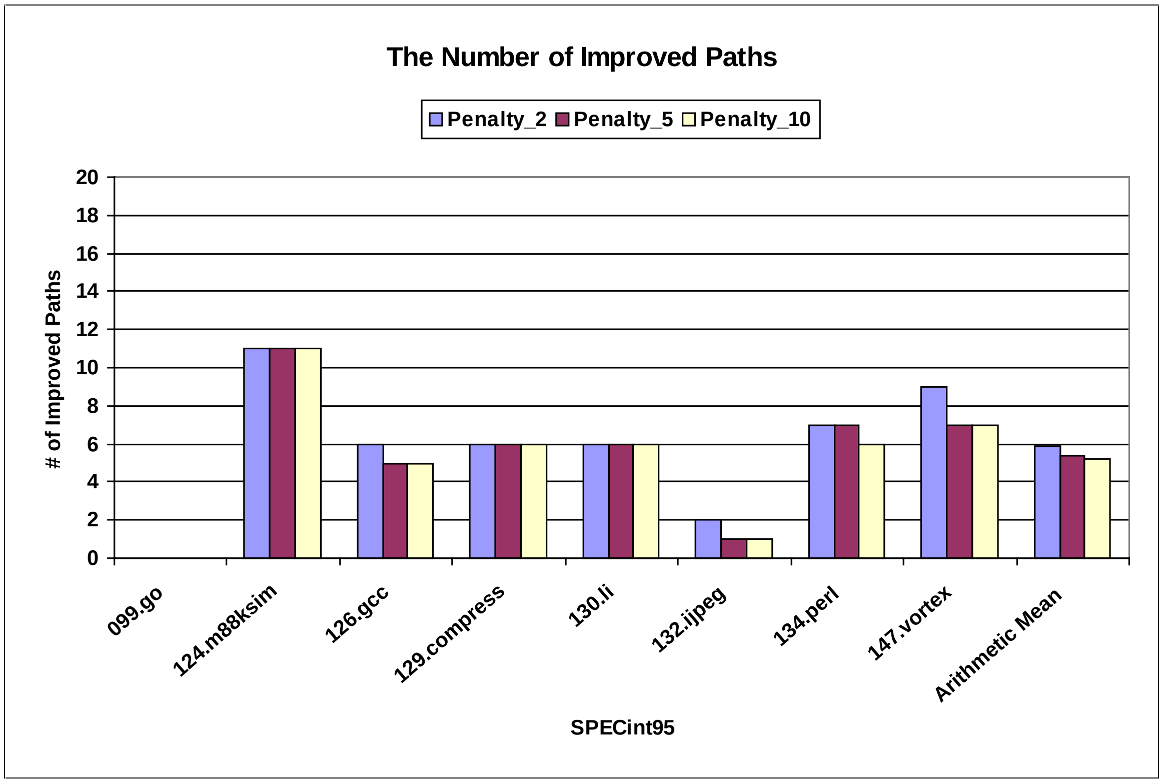


Figure 3.15 The number of improved paths using branch misprediction penalties of 2, 5, and 10 cycles. (Note that 099.go has no improved paths in all cases.)

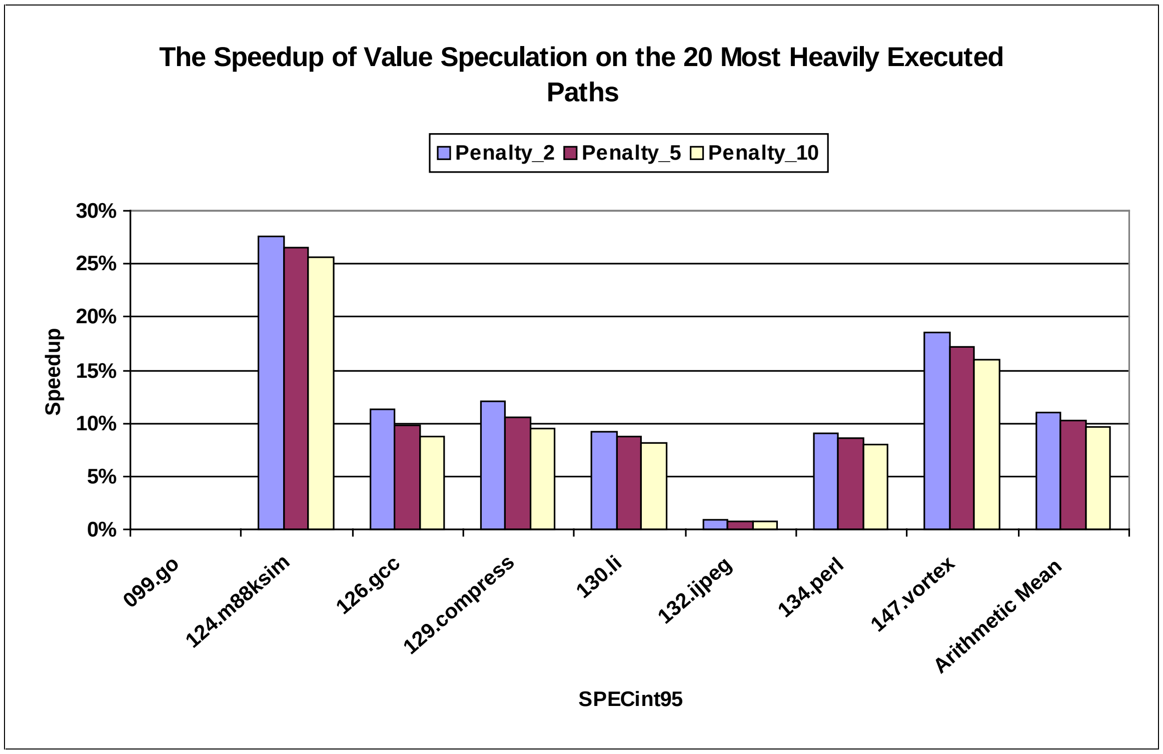


Figure 3.16 The speedup of value speculation on the 20 most heavily executed paths using branch misprediction penalties of 2, 5, and 10 cycles. (Note that 099.go has no speedups in all cases.)

3.5.3 Results of Selected Edges and Nodes

Knowing the maximal benefit from value speculation in Section 3.5.2, the optimal set of edges selected by the algorithm is examined in terms of the value prediction accuracy distribution and the location distribution. The location distributions and the opcodes of the selected producer and consumer operations are also presented.

The value prediction accuracy distribution of the selected edges using a 10-cycle branch misprediction penalty is shown in Figure 3.17. Surprisingly, 66% of the selected edges have value prediction accuracies over 99%. There are very few selected edges whose prediction accuracies are lower than 94%. The high penalties for value misprediction make the optimal edge selection algorithm select very highly predictable edges to break. This implies that the high value prediction accuracy threshold or the confidence mechanism [8] is necessary to reduce the impact of value misprediction.

The location distribution of the selected edges is shown in Figure 3.18. The locations in the data dependence graph are normalized to the original length of the data dependence graph. Zero indicates the top location and one indicates the bottom location in the data dependence graph. Each bar in Figure 3.18 shows the percentage of edges that cross the specific location in the data dependence graph. In Figure 3.18, 21.27% of edges cross the 0.4 normalized location, producing the most frequent crossing area. The selected edges tend to span the middle location of the data dependence graph, but more rarely cross the locations toward the top or the bottom. Intuitively, breaking edges that cross the middle location obtains large cycle savings by splitting the data dependence graph into half.

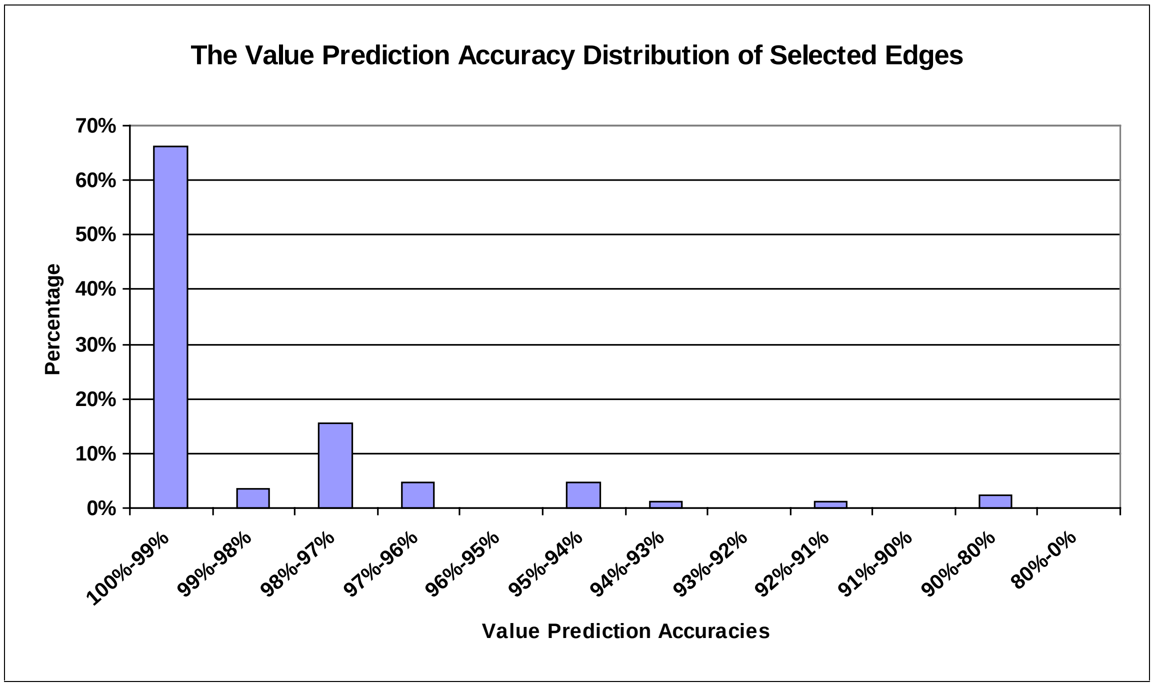


Figure 3.17 The value prediction accuracy distribution of the selected edges (using a 10-cycle branch misprediction penalty).

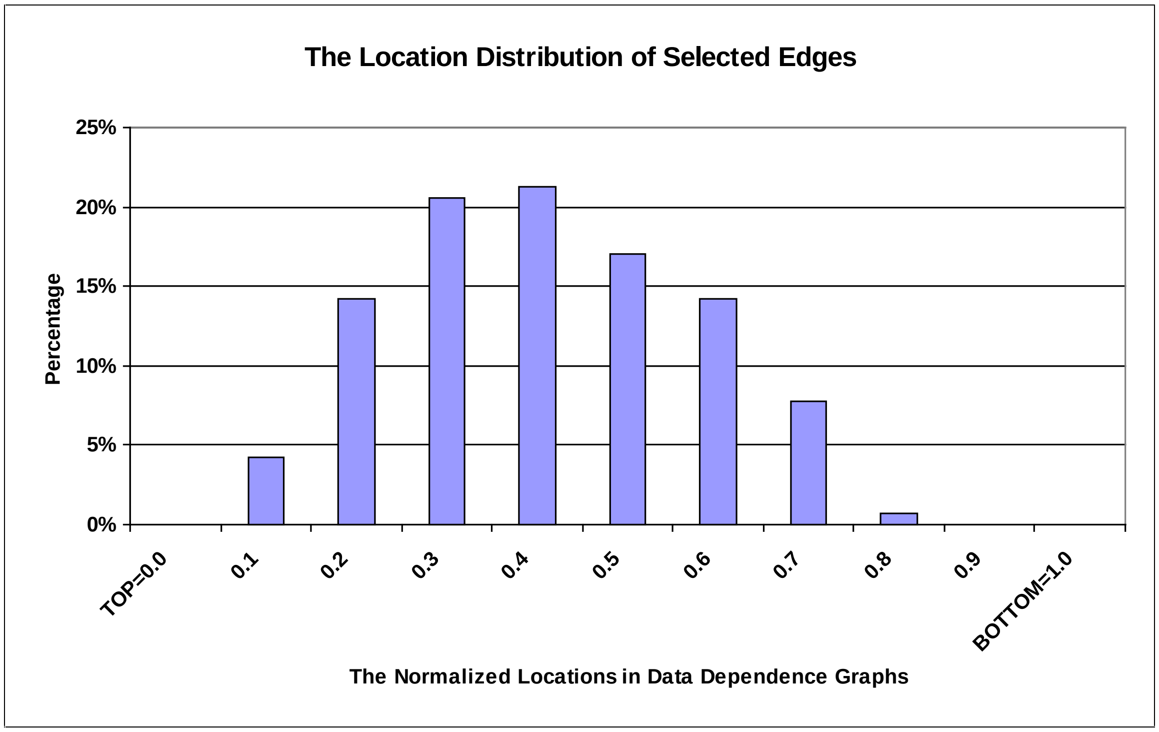


Figure 3.18 The location distribution of the selected edges (using a 10-cycle branch misprediction penalty).

After examining the locations of the selected edges, the location distributions of the selected producer and consumer operations are shown in Figures 3.19 and 3.20. Each bar in Figures 3.19 and 3.20 represents the percentage of operations that originally stay between two adjacent normalized locations in the data dependence graph. In Figure 3.19, the selected producer operations are likely to reside in the upper portion of the data dependence graph. 26.5% of the producer operations are between the 0.2 and 0.3 normalized locations, producing the most frequent crossing area. Conversely, the selected consumer operations appear toward the lower part of the data dependence graph. 21.68% of the consumer operations are between the 0.6 and 0.7 normalized locations, producing the most frequent crossing area. The different locations of the producer and consumer operations may explain why most of the selected edges cross the middle locations of the data dependence graph. From the results that the selected producer and consumer operations appear in many different locations, the heuristics of selecting operations based on only one specified location [24] may not obtain maximal benefits from value speculation.

The top five opcodes of the selected producer and consumer operations are shown in Table 3.2. The top two opcodes of the producer operations are *LOAD (WORD)* with 49.39% and *LOAD (BYTE)* with 13.25%. Because load operations have two-cycle latencies and other integer operations have one-cycle latencies, load operations are usually on the critical paths. Predicting load operations can often reduce the length of the critical path, so load operations are selected most frequently. For the consumer operations, *ADD*, *AND*, and *OR* are the top three opcodes with 27.71%, 15.66%, and 14.45%. The percentage differences are small for different consumer operations, when compared to the percentage difference of the top two producer operations. The results suggest that long-latency operations be good candidates for value prediction. However, choosing different operations to consume the predicted values does not matter significantly.

Table 3.2 The top five opcodes and percentages of the selected producer and consumer operations (using a 10-cycle branch misprediction penalty).

Rank	**Opcode and Percentage of Producer Operations**	**Opcode and Percentage of Consumer Operations**
1	LOAD (WORD) => 49.39%	ADD => 27.71%
2	LOAD (BYTE) => 13.25%	AND => 15.66%
3	AND => 9.63%	OR => 14.45%
4	ADD => 7.22%	LOAD (WORD) => 10.84%
5	MOVE => 6.02%	SUB => 8.43%

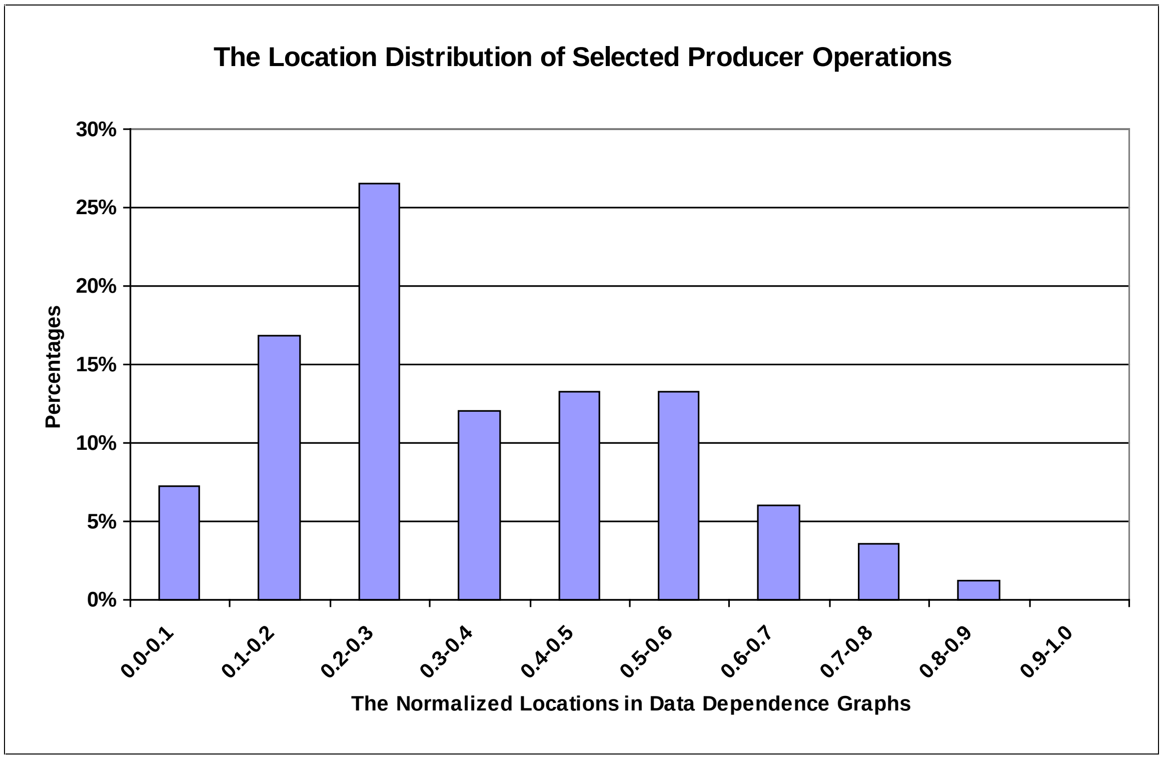


Figure 3.19 The location distribution of the selected producer operations in data dependence graphs (using a 10-cycle branch misprediction penalty).

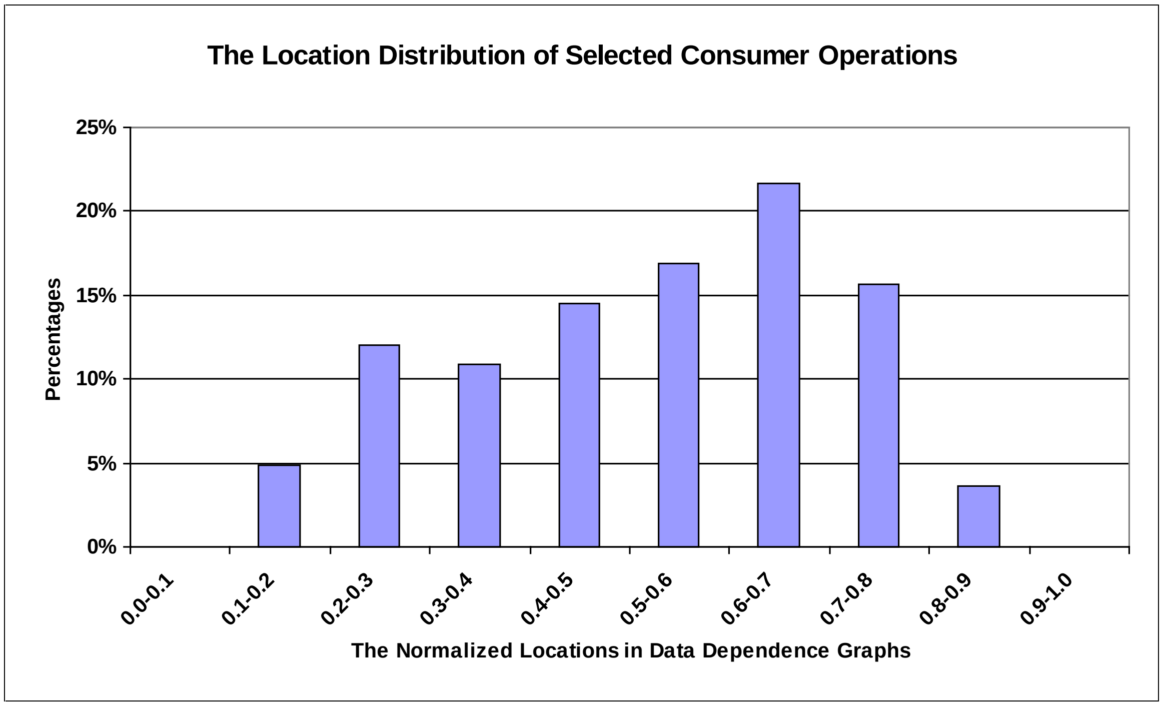


Figure 3.20 The location distribution of the selected consumer operations in data dependence graphs (using a 10-cycle branch misprediction penalty).

3.6 Summary

In this chapter, an optimal edge selection problem is proposed to model the value speculation techniques. This model helps us understand and improve the techniques for value speculation. Based on three properties observed from the optimal edge selection problem, an algorithm is designed to solve the optimal edge selection problem efficiently. The output of the optimal edge selection algorithm indicates an optimal set of edges that can be broken to obtain maximal benefits from value speculation. From the experimental results, the average speedup of value speculation is 9.61% on the 20 most heavily executed paths selected from each SPECint95 benchmark. Most of the selected edges have value prediction accuracies over 99%, so the impact of value misprediction is minimized. The location distributions of the selected edges and the opcodes of the selected producer operations provide insights to adjust the heuristic values used in the value speculation techniques.

In Chapter 4, the optimal edge selection algorithm is proposed to serve as a new compilation phase of benefit analysis to expose an optimal set of edges (dependences) to the dynamically-scheduled machines by using special bits in an instruction format for breaking flow dependences dynamically, or to the compiler for performing the VSS optimization statically. Thereby, the techniques for value speculation can provide the largest speedups for microprocessors.

Chapter 4

Compiler-Directed Edge Selection

It is a challenging task for a value speculation technique to select dependences to break to improve the performance of microprocessors. The reasons are many. Predicting operations with low predictability results in adverse effects on execution time due to frequent execution of recovery code (or frequent re-execution of operations that are affected by mispredicted values.) However, predicting operations with high predictability may not shorten execution time, because other data dependences may still exist in a program. Also, after predicting operations on a critical path, the newly created critical path may still limit program execution, resulting in critical path lengths that are not significantly shorter. Moreover, due to the nature of speculative execution, non-speculative and speculative operations compete for machine resources. As discussed in Section 3.1, the I-Cache, D-Cache, and branch predictor are all affected by speculative execution. If there are small benefits in terms of ideal execution cycles after applying value speculation, the penalties of I-Cache, D-Cache, and branch misprediction stalls will counteract the benefits, and the net speedup of value speculation will be negative.

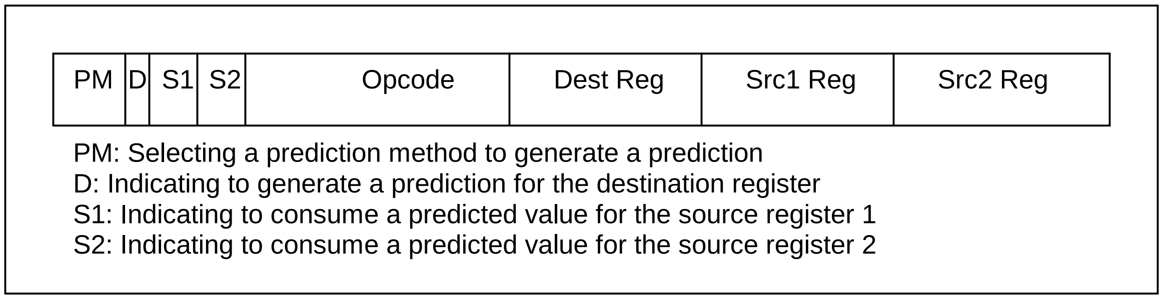
In this Chapter, we propose using compiler-directed edge selection to let the value speculation techniques know which dependences should be broken dynamically or statically. The compiler-directed edge selection serves as a new compilation phase of benefit analysis, which is an application derived from the optimal edge selection algorithm introduced in Chapter 3. Running the optimal edge selection algorithm finds a minimal set of edges (dependences) that the value speculation techniques can break to yield maximal benefits. The selected dependences are exposed to the dynamic hardware by using special fields in the instruction format or to the value speculation scheduler statically. To efficiently use the compiler-directed edge selection, employing value prediction accuracy thresholds can speed up the original optimal edge selection algorithm.

The remainder of this chapter is organized as follows. Section 4.1 describes the schemes of exposing compiler-directed edge selection to dynamically-scheduled and statically-scheduled machines. Section 4.2 introduces the heuristics to speed up the optimal edge selection algorithm. Section 4.3 presents experimental results of exposing compiler-directed edge selection to the VSS scheme [22], including the edge selection, the code size expansion, the register pressure, and the execution time speedup. Section 4.4 concludes this chapter.

4.1 Schemes of Exposing Compiler-Directed Edge Selection

To expose compiler-directed edge selection to dynamically-scheduled machines, special fields in an instruction are used to indicate the selected dependences that need to be broken. In [4], special directives in the opcodes are proposed to select specific value prediction methods, such as last value or stride. Figure 4.1 shows an instruction format that supports choosing a prediction method [4] and provides specific dependences for the value speculation techniques to break in dynamically-scheduled machines. Four new fields of *PM*, *D*, *S1*, and *S2* are added in an instruction. The PM field indicates which value predictor is used to generate a prediction. The bit width of the PM field depends on how many value predictors are available in the hardware. The D field is a 1-bit field to decide if the prediction needs to be made for this instruction. The S1 and S2 fields are 1-bit fields that determine to consume the predicted values for value-speculative execution, if the actual results of source registers are not ready. The new instruction format alleviates the burden for the hardware to dynamically decide which operation to predict and which operation to consume a predicted value. Previously, the instruction selection is based either on the confident level of predictions or on the critical path information of instructions [8], [9].

For statically-scheduled machines, the selected edges (dependences) are directly fed to the compiler to perform the VSS optimization [24]. The value speculation scheduler breaks the selected dependences by inserting predicting operations (LDPRED), generates recovery code, and schedules operations aggressively.



PM: Selecting a prediction method to generate a prediction
D: Indicating to generate a prediction for the destination register
S1: Indicating to consume a predicted value for the source register 1
S2: Indicating to consume a predicted value for the source register 2

Figure 4.1 An instruction format that can choose a prediction method and specify dependences for value speculation.

4.2 Heuristics Applied to the Optimal Edge Selection Algorithm

To efficiently use the compiler-directed edge selection, the running time of the optimal edge selection algorithm shown in Figure 3.9 must be reduced. Heuristics of value prediction accuracy thresholds are proposed to speed up solving the optimal edge selection problem. In Step 1 of the algorithm shown in Figure 3.9, the value prediction accuracy thresholds can be used to filter out more candidate edges. Five different value prediction accuracy thresholds of 80%, 85%, 90%, 95%, and 99% were experimented for the 20 most heavily executed paths in each SPECint95 benchmark.

Figure 4.2 shows the normalized number of edge selections that are tried by the original algorithm and the algorithms with five different thresholds. When increasing the value prediction accuracy threshold, the number of selections decreases by a certain amount. In 124.m88ksim, the thresholds of 80%, 85%, 90%, and 95% cannot filter out edges effectively; only the threshold of 99% reduces a lot of combinations, because most operations in 124.m88ksim are very highly predictable. In 099.go and 132.ijpeg, applying all value prediction accuracy thresholds yields very few or no selections, because operations in these benchmarks have very low predictability. In 126.gcc, 129.compress, 130.li, 134.perl, and 147.vortex, the value prediction accuracy thresholds of 90%, 95%, and 99% effectively reduce more than 80% of combinations compared to the original number of selections. For all benchmarks, the algorithm with the value prediction accuracy threshold of 99% yields the fewest combinations.

Figure 4.3 shows the normalized maximal benefit from value speculation that is found by the original algorithm and the algorithms with five different value prediction accuracy thresholds for the 20 most heavily executed paths in each SPECint95 benchmark. The algorithm with the value prediction accuracy threshold of 80% yields the same maximal benefit as the one found by the original algorithm for all SPECint95 benchmarks. Using the thresholds of 85% and 90%, the algorithm gets the same maximal benefit for all benchmarks except for 147.vortex. In 126.gcc, 129.compress, and 147.vortex, using the threshold of 95% reduces the maximal benefit by 3.5%, 29%, and 13% respectively. In Figure 4.2, the threshold of 99% can reduce the most running time of the original algorithm, but the corresponding maximal benefit drops significantly for most of the benchmarks as shown in Figure 4.3. From the results of Figures 4.2 and 4.3, the value prediction accuracy threshold of 90% is a good heuristic value that can be applied to the original optimal edge selection algorithm to speed up the analysis by more than 80% for most of the SPECint95 benchmarks, and still get benefits close to the original maximal benefits.

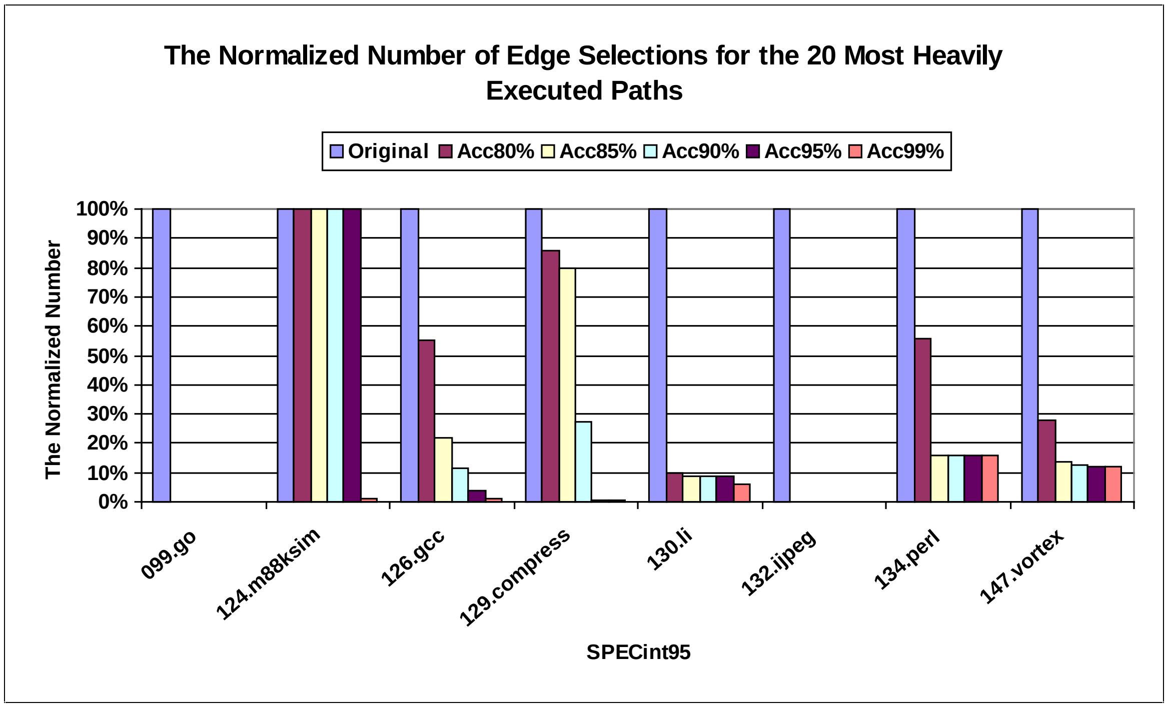


Figure 4.2 The normalized number of edge selections that are tried by the original algorithm and the algorithms with five value prediction accuracy thresholds for the 20 most heavily executed paths in SPECint95.

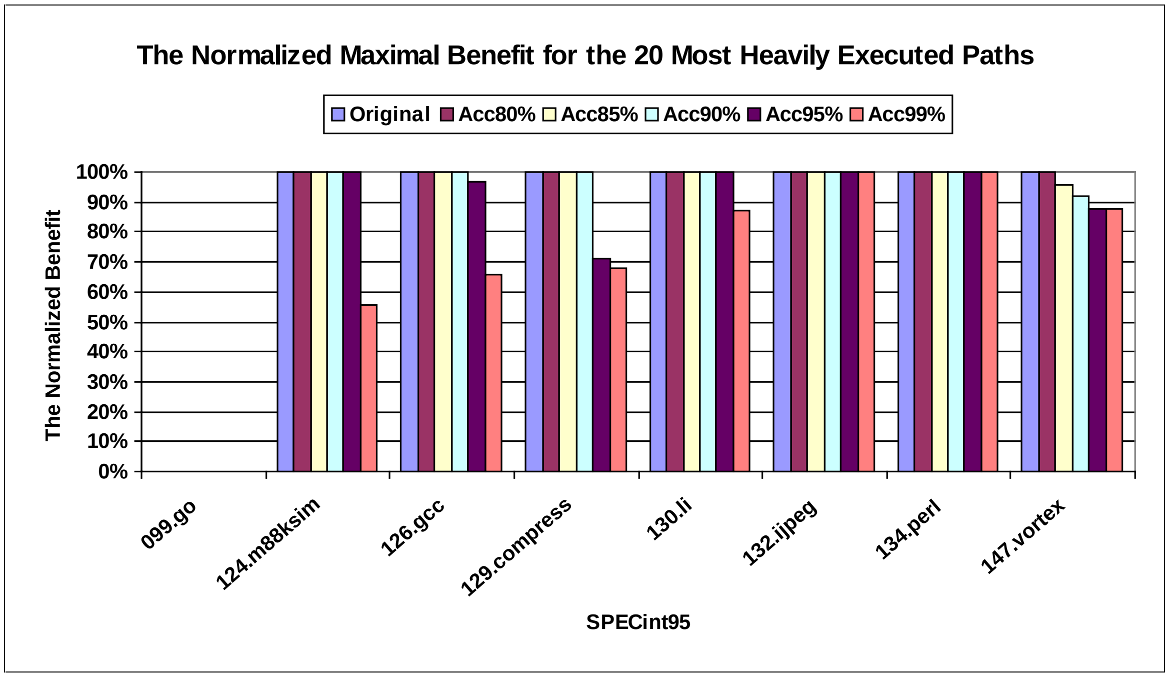


Figure 4.3 The normalized maximal benefit using the original algorithm and the algorithms with five value prediction accuracy thresholds for the 20 most heavily executed paths in SPEint95. (Note that 099.go has no benefits in all cases.)

4.3 Experimental Results

Several experimental results of exposing compiler-directed edge selection to the VSS optimization are presented in detail, including the edge selection in Section 4.3.1, the code size expansion in Section 4.3.2, the register pressure in Section 4.3.3, and the execution time speedup in Section 4.3.4.

4.3.1 Edge Selection

In the experiments, the optimal edge selection algorithm with the value prediction accuracy threshold of 90% was performed on the 20 most heavily executed paths selected from each SPECint95 benchmark. Table 4.1 shows the results of the maximal benefit, the number of selected edges, the number of selected producer operations, and the number of selected consumer operations. The maximal benefit is the average percentage of critical path reduction for the 20 most heavily executed paths in each benchmark. In Table 4.1, 124.m88ksim and 147.vortex have the highest benefits of 25.57% and 14.69%. In 126.gcc, 129.compress, 130.li, and 134.perl, the maximal benefits are significant with 8%. 099.go and 132.ijpeg are not good candidates for value speculation, because their benefits are negligible.

In Table 4.1, the number of the selected edges indicates how many dependences are to be broken via the value speculation techniques. The number of the selected producer operations is the number of predictions that the value predictor generates, and the number of the selected consumer operations is the number of operations that consume predicted values for the source operands. For all programs except for 124.m88ksim and 129.compress, the numbers of edges, producer operations, and consumer operations are the same. This means that each predicted operation feeds to only one consumer operation, and each consumer operation receives only one predicted value for value-speculative execution. In 124.m88ksim and 129.compress, some predicted values are used by more than one consumer operations, and some consumer operations use more than one predicted values for the source operands.

Table 4.1 Results of the optimal edge selection algorithm with the value prediction accuracy threshold of 90% on the 20 most heavily executed paths in SPECint95.

SPECint95	**The Maximal Benefit**	**# of Selected Edges**	**# of Producer Operations**	**# of Consumer Operations**
099.go	0%	0	0	0
124.m88ksim	25.57%	33	30	28
126.gcc	8.81%	8	8	8
129.compress	9.49%	15	11	14
130.li	8.15%	6	6	6
132.ijpeg	0.81%	1	1	1
134.perl	8.04%	9	9	9
147.vortex	14.69%	11	11	11
Average	**9%**	**10.375**	**9.5**	**9.625**

For further experiments on the VSS optimization, the SPECint95 benchmarks except for 099.go and 132.ijpeg were chosen. The selected edges (dependences) were exposed to the compiler to perform VSS for a 16-issue VLIW machine model based on the Hewlett-Packard Laboratories HPL-PD architecture [20]. All operations have a one-cycle latency except for load (two cycles), floating-point add (two cycles), floating-point subtract (two cycles), floating-point multiply (threes cycles), and floating-point divide (three cycles). The SPECint95 programs were compiled with classic optimizations by the IMPACT compiler from the University of Illinois [18] and converted to the Rebel textual intermediate representation by the Elcor compiler from Hewlett-Packard Laboratories [19]. Then, the LEGO compiler [17] scheduled base code (without the VSS optimization) and VSS-optimized code.

Table 4.2 The code size of base code and VSS-optimized code in SPECint95. (The unit is the number of single operations.)

SPECint95	Base Code	VSS-Optimized Code	Difference (VSS – Base)	Ratio (VSS / Base)
124.m88ksim	46,496	47,165	669	1.014
126.gcc	500,602	500,957	355	1.000
129.compress	1,773	2,213	440	1.248
130.li	14,982	15,191	209	1.014
134.perl	92,253	92,876	623	1.006
147.vortex	212,471	212,785	314	1.001
Average	**144,762**	**145,197**	**435**	**1.048**

4.3.2 Code Size Expansion

Table 4.2 shows the code size of base code and VSS-optimized code. The unit is the number of single operations in a program. The difference and the ratio between VSS-optimized code and base code are presented in columns 4 and 5 of Table 4.2. After applying VSS, the code size increases, because recovery code and new operations, such as LDPRED, UDPRED, and BNE [24], are generated in VSS-optimized code. The difference of VSS-optimized code and base code is averaging 435 operations, and the code size ratio is 1.048 on an average. 124.m88ksim has the largest code size difference of 669 operations, because there are the most selected edges to be broken as shown in Table 4.1. 134.perl is second with 623 additional operations in VSS-optimized code. 130.li has the fewest additional operations, because only six edges are selected in Table

4.1. If the code size expansion is huge, the performance of the I-Cache will be degraded and the benefits from value speculation will be affected.

4.3.3 Register Pressure

Table 4.3 presents the register usage in the procedures that are different between base code and VSS-optimized code. In general, because the VSS scheme creates recovery code and uses new registers to store the predicted values, the register usage of VSS-optimized code is higher than that of base code.

Table 4.3 The register usage in the procedures that are different between base code and VSS-optimized code.

	Base Code			VSS-Optimized Code		
Register Usage	**Average**	**Maximum**	**Spilled**	**Average**	**Maximum**	**Spilled**
124.m88ksim	12.26	37	0	13.19	37	0
126.gcc	21.75	52	0	22.5	45	0
129.compress	9.33	21	0	10	25	0
130.li	6.22	18	0	6.31	18	0
134.perl	15.82	128	1	15.94	128	1
147.vortex	14.49	65	0	14.61	65	0
Overall	**13.31**	**128**	**1**	**13.76**	**128**	**1**

For all programs, VSS-optimized code uses averaging 13.76 registers that are higher than 13.31 registers for base code. In our VLIW machine model, there are 128 integer registers. If the register allocator cannot assign 128 architectural registers to all virtual registers used in a procedure, the spilling and filling code will be introduced. In 134.perl, base code and VSS-optimized code contain one spilled register. For the other benchmarks, up to 65 registers are enough to assign all virtual registers in base code and VSS-optimized code. From the results of Table 4.3, VSS-optimized code does not increase the register pressure very much. However, in one procedure of 126.gcc, VSS-optimized code actually requires fewer registers than base code does. Because the ILP transformation via VSS breaks the critical path and may shorten the register lifetime, fewer architectural registers are required to assign all virtual registers in the procedure of VSS-optimized code in 126.gcc.

4.3.4 Execution Time Speedup

Trace simulation was performed for evaluating the speedup of VSS-optimized code over base code on nine machine models, which are composed of functional blocks shown in Table 4.4. In Table 4.4, there are four functional blocks: *execution*, *I-Cache*, *D-Cache*, and *branch predictor* blocks. The execution block is capable of issuing and executing 16 operations per cycle with 16 universal functional units. The I-Cache block is a 64k-byte compressed I-Cache with two banks [33]. The D-Cache block is a 4-way cache with 64k-byte data storage. The branch predictor block employs multi-way branch prediction [34], [35] with 2^{14} entries in the branch prediction table (BPT) and 2^{14} entries in the branch target buffer (BTB). The branch misprediction stalls are 2, 5, or 10 cycles.

Figure 4.4 shows the execution time speedup of VSS-optimized code over base code using nine machine models. The first model is composed of the execution block, and represents an ideal model that calculates pure execution cycles in the pipeline. The next five models consist of the execution and I-Cache, execution and D-Cache, execution and branch (2-cycle stalls), execution and branch (5-cycle stalls), and execution and branch (10-cycle stalls). The last three models represent realistic models that contain the execution, I-Cache, D-Cache, and branch predictor with 2-, 5-, or 10-cycle stalls.

Table 4.4 Functional blocks used for composing different machine models.

Blocks	Specification
Execution	Dispatch/issue/retire bandwidth: 16 Universal functional units: 16 Operation latencies are described in Section 4.3.1.
I-Cache	Compressed (zero-nop) and two banks with 64k bytes [33] Line size = 16 operations (each bank) Miss penalty = 12 cycles
D-Cache	Size/assoc./repl. = 64kB/4-way/LRU Line size = 32 bytes Miss penalty = 14 cycles
Branch Predictor	Multi-way branch prediction [34], [35] Branch prediction table (BPT) = 2^{14} entries Branch target buffer (BTB) entry/assoc./repl. = 2^{14}/8-way/LRU Branch misprediction stalls = 2, 5, 10 cycles

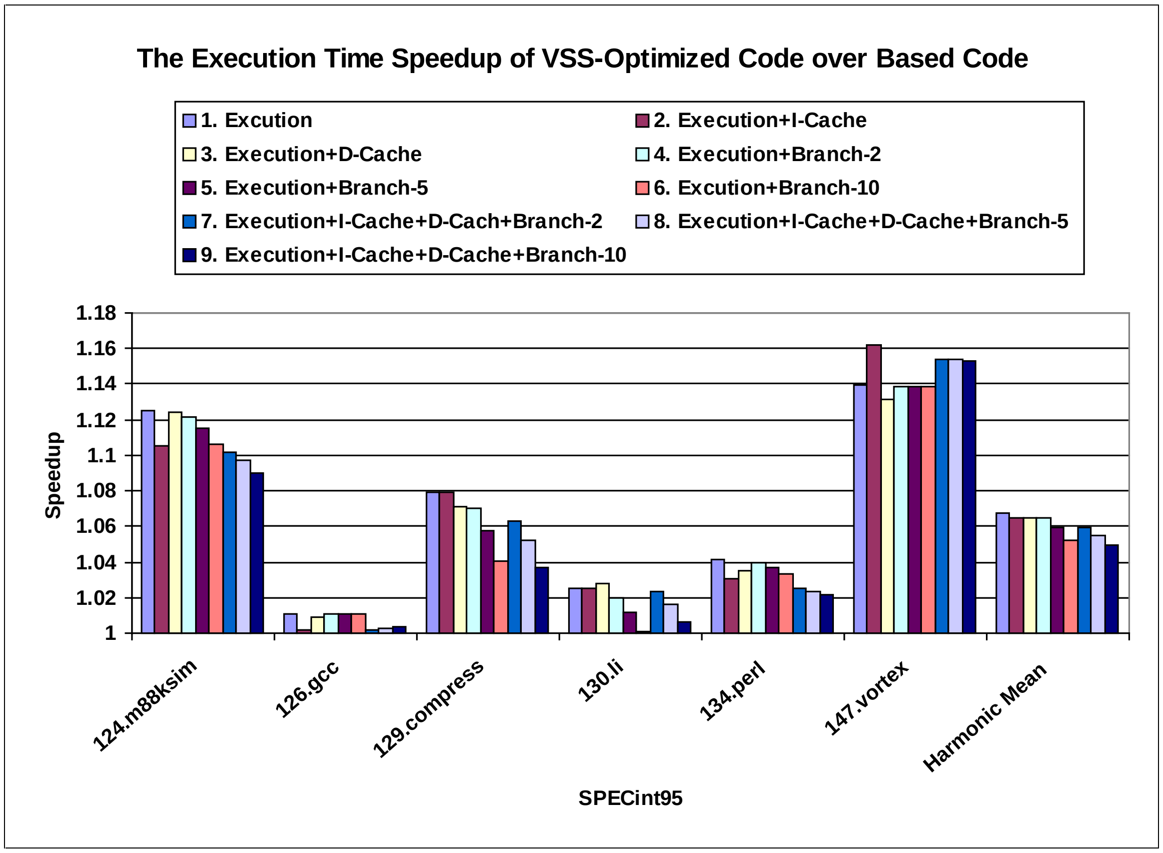


Figure 4.4 The execution time speedup of VSS-optimized code over based code using nine machine models.

In Figure 4.4, the speedups of VSS-optimized code over base code are positive under all machine models. 147.vortex gets the highest speedups ranging from 13.5% to 16%. 124.m88ksim is second with speedups between 9% and 12.5%. 129.compress has the third highest speedups ranging from 3.8% to 8%. 134.perl and 130.li have moderate speedups, less than 4 % or 3%. 126.gcc has the lowest speedup of 1% or smaller. As shown in Table 4.1, although the maximal benefits found by the optimal edge selection algorithm for the 20 most heavily executed paths are more than 8% in the selected SPECint95 benchmarks, the execution time speedups of VSS-optimized code over base code are different in Figure 4.4. 126.gcc has small speedups, because its 20 most heavily executed paths account for less than 10% of dynamic execution. In other SPECint95 benchmarks, the 20 paths account for more than 50% of execution time, but their speedups of value speculation are still not the same. In 124.m88ksim, 129.compress, and 147.vortex, the scheduler does a good job of scheduling VSS-optimized code. However, after applying VSS to 130.li and 134.perl, the scheduler does not fully exploit the exposed ILP, so their speedups are small.

When comparing the speedups by using different machine models, Model 1 gets the highest speedups in all programs except for 147.vortex. In 147.vortex, using Models 2, 7, 8, and 9 with the I-Cache has higher speedups than using Model 1. The reason is that the VSS-optimized code changes the formation of basic blocks and exhibits better performance of the I-Cache in 147.vortex. When using Model 3 with the D-Cache, the speedups decrease in all programs except for 130.li. In general, the VSS optimization generates more load and store operations to access the D-Cache, and the increased D-Cache stalls reduce the speedups of value speculation. When branch predictors are incorporated into Models 4, 5, and 6, the corresponding speedups decrease with increased branch misprediction stalls. For most cases, VSS-optimized code experiences more branch misprediction stalls than base code does. Using the most realistic machine model (Model 9) still obtains significant speedups of up to 15%, and 5% on a harmonic mean.

Table 4.5 shows the execution time breakdown of base code and VSS-optimized code when using Model 9. In Table 4.5, the pipeline stalls account for the largest portion of execution time. VSS-optimized code always has fewer pipeline stalls than base code does. In many cases, VSS-optimized code increases the I-Cache, D-Cache, and branch misprediction stalls. However, in some benchmarks, VSS-optimized code has fewer I-Cache, D-Cache, or branch misprediction stalls that are highlighted by bold fonts in Table 4.5. Especially for investigating the performance of the multi-way branch predictor, the branch misprediction rates of base code and VSS-optimized code are shown in Figure 4.5. The average branch misprediction rate is 5.5% for base code and 5.6% for VSS-optimized code. In three benchmarks of 124.m88ksim, 126.gcc, and 147.vortex, VSS-optimized code has lower branch misprediction rates, while in the other three benchmarks of 129.compress, 130.li, and 134.perl, VSS-optimized code has higher branch misprediction rates. The lower branch misprediction rates in VSS-optimized code do not always indicate that there are fewer branch misprediction stalls, e.g., 124.m88ksim in Table 4.5. The reason is that VSS-optimized code contains more branches than base code does, so the total branch misprediction stalls of VSS-optimized code are still higher than those of base code. The statistics of multi-way branches in base code and VSS-optimized code are shown in Table 4.6. Note that each multi-op can have up to 16 branches in the 16-issue VLIW machine model.

Table 4.5 The execution time breakdown of base code and VSS-optimized code by using Model 9. (Bold fonts indicate that VSS-optimized code performs better than base code does.)

SPECint95		Pipeline Stalls	I-Cache Stalls	D-Cache Stalls	Branch Stalls
124.m88ksim	**Base**	68,072,621	8,974,476	85,706	5,002,520
	VSS	**60,500,902**	9,181,224	123,689	5,533,940
126.gcc	**Base**	591,322,161	232,412,436	51,586,022	257,558,640
	VSS	**585,128,808**	236,914,320	51,951,316	**255,004,760**
129.compress	**Base**	18,756,881	1,056	1,989,003	3,432,190
	VSS	**17,381,298**	1,368	1,989,118	3,950,480
130.li	**Base**	109,920,236	4,776	23,544,167	23,913,270
	VSS	**107,172,257**	4,848	**22,622,106**	26,562,110
134.perl	**Base**	793,170,935	105,091,332	142,749,914	126,116,250
	VSS	**761,828,702**	110,032,188	**142,676,454**	128,172,090
147.vortex	**Base**	1,007,199,307	413,578,260	64,072,286	64,574,530
	VSS	**884,164,188**	**338,881,688**	**63,008,854**	**57,465,640**

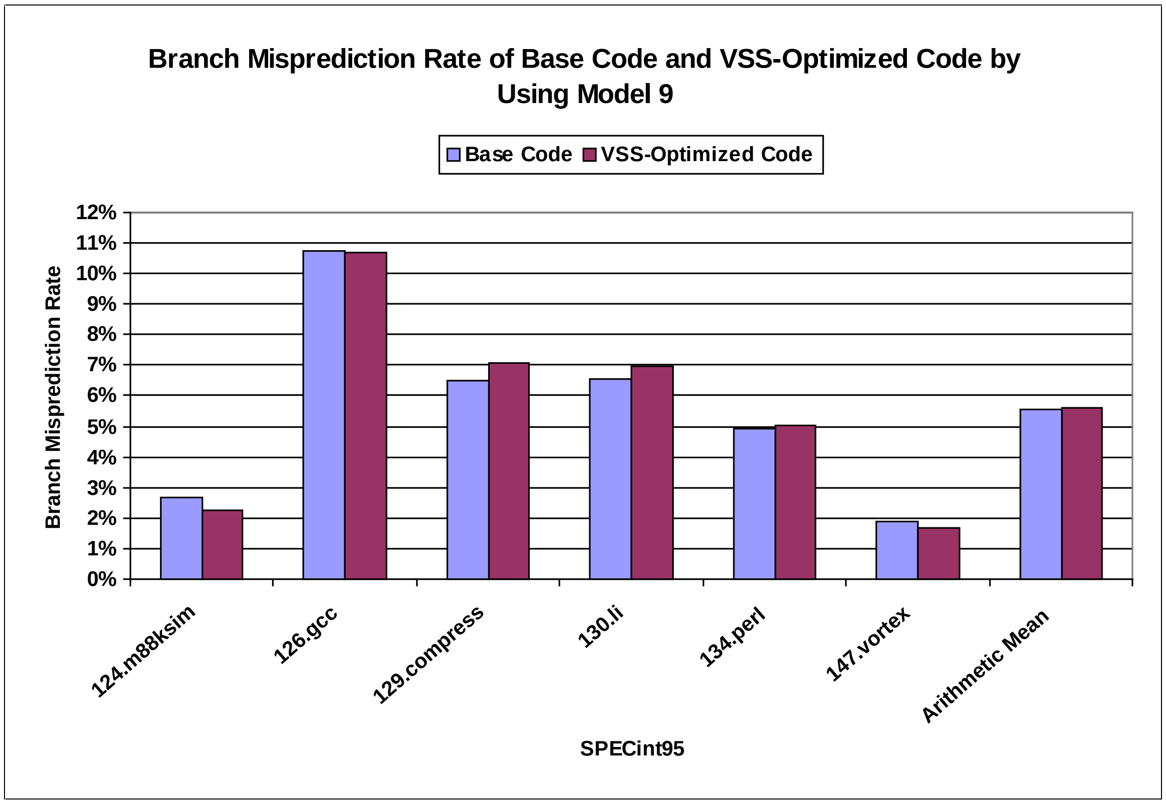


Figure 4.5 Branch misprediction rates of base code and VSS-optimized code by using Model 9.

Table 4.6 Statistics of multi-way branches in base code and VSS-optimized code. Each data represents the number of multi-ops that contains a certain number of single branches from 1 to 16.

	124.m88ksim		126.gcc		129.compress		130.li		134.perl		147.vortex	
#	**Base**	**VSS**	**Base**	**VSS**	**Base**	**VSS**	**Base**	**VSS**	**Base**	**VSS**	**Base**	**VSS**
1	6144	6210	71518	71540	233	264	2919	2951	13089	13140	27714	27779
2	1535	1549	24390	24392	78	87	590	599	4026	4047	8422	8427
3	363	364	4240	4248	7	9	94	94	790	800	1598	1598
4	160	164	1866	1867	3	3	37	38	285	291	567	569
5	82	83	957	961	3	5	16	16	158	160	182	182
6	29	29	626	626	0	1	10	10	77	79	61	61
7	52	53	643	643	1	1	7	7	60	62	45	45
8	64	64	690	691	0	0	14	14	86	90	62	63
9	0	0	22	22	0	0	0	0	3	3	2	2
10	1	1	14	14	0	0	0	0	1	2	0	0
11	0	0	12	12	0	0	0	0	0	0	1	1
12	0	0	2	2	0	0	0	0	2	3	2	2
13	0	0	4	4	0	0	0	0	2	3	0	0
14	0	0	4	4	0	0	0	0	1	1	0	0
15	1	1	4	4	0	0	0	0	1	1	0	0
16	1	1	20	20	0	0	0	0	18	17	0	0
Sum	**8432**	**8519**	**105012**	**105050**	**325**	**370**	**3687**	**3729**	**18599**	**18699**	**38656**	**38729**

4.4 Summary

In this chapter, the schemes of exposing compiler-directed edge selection are proposed for the value speculation techniques in dynamically-scheduled and statically-scheduled machines. For dynamically-scheduled machines, the new instruction format that contains four new fields is designed to select a prediction method and expose the specific dependences to the hardware. The instruction format alleviates the burden for the hardware to dynamically decide which dependences to break via value speculation. For statically-scheduled machines, the selected dependences are directly fed to the value speculation scheduler. The experimental results of exposing compiler-directed edge selection to the VSS optimization are presented in detail, including the edge selection, the code size expansion, the register pressure, and the execution time speedup. Speedups of up to 15% and averaging 5% have been shown on a realistic machine model for optimizing the 20 most heavily executed paths in the SPECint95 benchmarks.

Chapter 5

Software-Only Value Speculation Scheduling

Value prediction [2], [3], [6] is an interesting research topic that has been investigated since 1996. Researchers and computer architects try to exploit the predictability for the values generated by register-writing operations to improve the performance of microprocessors. Techniques for value prediction and value speculation have been proposed as hardware-managed mechanisms [2], [3], [5], [8], [9], combined hardware and compiler synergies [4], [22], [24], [25], [26], or pure software schemes [23], [26]. The pure software techniques have the advantage of being applicable to existing microprocessors without adding new value prediction hardware and modifying processor pipelines to support value-speculative execution. In this chapter, we propose *software-only value speculation scheduling* (*SVSS*) to shorten program execution time, and investigate the performance of software static stride value predictors. The software static stride value predictor is chosen because it is very simple and requires at most one operation to generate predicted values. Instead of using filling and spilling code [23] to preserve register values for implementing software static stride value predictors, we propose using global registers [32] to reduce the overhead. Surprisingly, from the experimental results, the software static stride value predictor obtains the prediction accuracy of 95.47% comparable to 95.81% by using the hardware stride two-delta value predictor [10], [13] for predictable operations (whose prediction accuracies are higher than 50%) in the SPECint95 benchmarks. From the results of stride profiling, most of the predictable operations have very few distinct stride values. 0, 1, 4, -1, and -4 are the most frequently occurring stride values.

Having a certain amount of predictability of operations, the benefit analysis is performed to know which dependences should be broken to obtain maximal benefits from value speculation. Analyzing benefits for the 20 most heavily executed paths in each SPECint95 benchmark shows that the average critical path reduction is 9.43%. From the simulation for a VLIW machine model with the I-Cache, D-Cache, and multi-way branch predictor that has five-cycle stalls, the execution time speedup of SVSS-optimized code over base code has shown to be encouraging with up to 15%, and averaging 4%. These results are based on a configuration of up to 30 global registers available for implementing software static stride value predictors. Modern microprocessors, MIPS R10000 [29], Alpha 21264 [30], and Intel Itanium [27], [28], [31], [41] have 64, 80, and 128 physical integer (or general) registers, but few registers are both logical (or architectural) and global for the compiler usage. Intel Itanium has 32 global registers where 18 registers (r14-r31) are scratch registers and may be utilized experimentally for the SVSS optimization.

The remainder of this chapter is organized as follows. Section 5.1 introduces software-only value speculation scheduling. Section 5.2 presents the design and analysis of software static stride value predictors. Section 5.3 presents the experimental results. Section 5.4 concludes this chapter.

5.1 Software-Only Value Speculation Scheduling

The compiler optimization using software static stride value predictors is an ILP transformation that inserts software static stride value predictions to break flow (true) dependences in a program. This optimization is called *software-only value speculation scheduling (SVSS)*. Compared to the traditional scheduler that must honor all true dependences among operations to form a correct schedule, the value speculation scheduler can break true dependences and speculatively schedule value-dependent operations aggressively. The scheduler inserts a predicting operation to generate a predicted value and a verifying operation to compare the predicted value with the actual result. In the case of value misprediction, compiler-generated recovery code is used to re-execute operations that are affected by incorrect predictions.

Figure 5.1 shows examples of ILP transformation via value speculation scheduling (VSS) introduced in Chapter 2 and software-only value speculation scheduling (SVSS). Figure 5.1(a) lists a sequence of operations that are taken from 129.compress in the SPECint95 benchmarks. Figure 5.1(b) presents new code after applying VSS, and Figure 5.1(c) lists new code after applying SVSS. The difference between VSS and SVSS is that the former has an explicit ISA to manage value prediction hardware, but the latter uses simple ALU operations (ADD and MOVE) to emulate value prediction. The advantage of SVSS over VSS is that the value prediction hardware is not required and SVSS can be applied to existing microprocessors.

(a) Original code	**(b) New code after applying VSS**	**(c) New code after applying SVSS**
17: ADD R8 ← Label, 0	17: ADD R8 ← Label, 0	17: ADD R8 ← Label, 0
7: AND R2 ← R26, 255	7: AND R2 ← R26, 255	7: AND R2 ← R26, 255
8: LW **R4** ← 0(R8)	8: LW **R4** ← 0(R8)	8: LW **R4** ← 0(R8)
10: ADD R3 ← **R4**, 1	**// load prediction from hardware**	**// calculate software static stride prediction**
11: SW 0(R8) ← R3	**21: LDPRED R9 ← index**	**21: ADD R9 ← R9, stride**
12: SW 0(**R4**) ← R2	10: ADD R3 ← **R9**, 1	10: ADD R3 ← **R9**, 1
Next:	11: SW 0(R8) ← R3	11: SW 0(R8) ← R3
	12: SW 0(**R4**) ← R2	12: SW 0(**R4**) ← R2
	// verify prediction	**// verify prediction**
	22: BNE Recovery R9, R4	**22: BNE Recovery R9, R4**
	Next:	Next: ......
	Recovery:	**Recovery:**
	// update hardware predictor	**// update software static stride predictor**
	23: UDPRED R4, index	**23: MOVE R9 ← R4**
	10': ADD R3 ← **R4**, 1	10': ADD R3 ← **R4**, 1
	11': SW 0(R8) ← R3	11': SW 0(R8) ← R3
	12': SW 0(**R4**) ← R2	12': SW 0(**R4**) ← R2
	24: JMP Next	24: JMP Next

Figure 5.1 Examples of ILP transformation via VSS and SVSS.

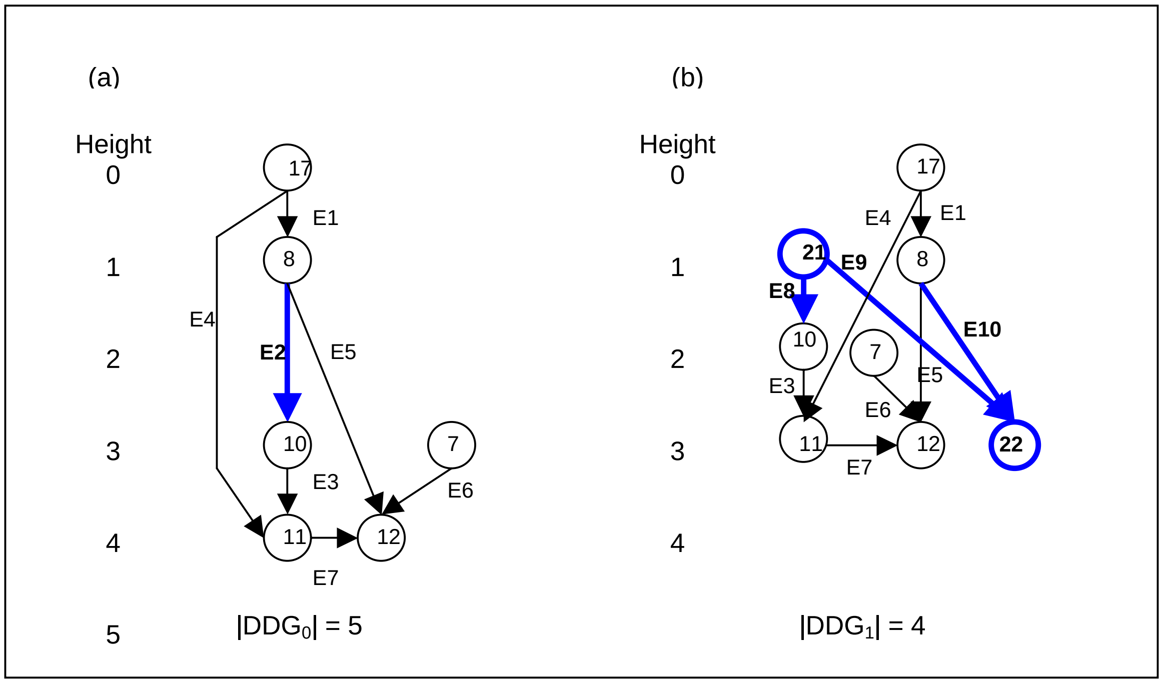


Figure 5.2 (a) The data dependence graph for code in Figure 5.1(a). (b) The data dependence graph for code in Figures 5.1(b) or 5.1(c). Thick edges and thick-circled nodes are deleted or created by VSS or SVSS.

In Figure 5.1(b), after applying VSS, three new operations are inserted: an LDPRED, operation 21, generating a prediction from a hardware value predictor, a BNE, operation 22, verifying the prediction with the correct result, and a UDPRED, operation 23, correcting the state of the hardware value predictor. Similarly, in Figure 5.1(c), after applying SVSS, three new operations are inserted: an ADD, operation 21, providing a prediction to register R9 by adding a static stride to its previous result, a BNE, operation 22, verifying the prediction with the correct result, and a MOVE, operation 23, storing the correct result R4 to R9. For VSS and SVSS, the compiler is responsible for generating recovery code to redirect program execution after value misprediction. After applying VSS and SVSS, the original data dependence graph shown in Figure 5.2(a) is collapsed, and the resultant data dependence graph shown in Figure 5.2(b) has a shorter length with more instruction-level parallelism available for the scheduler to exploit statically [22], [23] or dynamically [26].

To improve the performance of microprocessors via SVSS, two aspects should be paid attention to: obtaining high prediction accuracies through the use of software static stride value predictors and performing benefit analysis for SVSS. To obtain high prediction accuracies, the optimum static stride value needs to be determined for each predicted operation. The benefit analysis is necessary to know how many optimization opportunities for SVSS exist in a program and where SVSS should be applied. As presented in Chapters 3 and 4, the benefit analysis is performed by solving an optimal edge selection problem in a data dependence graph. The optimal edge selection problem involves finding an optimal set of edges (dependences) to break to obtain maximal benefits. After determining which dependences should be broken, the following tasks for the compiler become simple: inserting the software static stride value prediction, breaking the dependence, generating recovery code, and scheduling operations aggressively with newly exposed instruction-level parallelism.

5.2 Design and Analysis of Software Static Stride Value Predictors

As shown in Figure 5.1(c), software static stride value predictors use an ADD operation to generate a prediction by adding a constant value (static stride) to a register. If the static stride value equals 0, the ADD operation can be eliminated. For the best performance, the static stride values for operations are obtained through program profiling [4], [7].

To reduce the profiling overhead to determine static stride values, two profiling steps are performed: *value prediction accuracy profiling* and *stride profiling*. For the value prediction accuracy profiling, operations are profiled using hardware stride value predictors [4], [10], [13]. The first profile result contains the prediction accuracies of operations using hardware stride value predictors. For the stride profiling, only operations whose prediction accuracies are higher than 50% (from the first profile result) are profiled again. The second profile results show how many different stride values occur and how many times each distinct stride value appears for each profiled operation. The two profiling steps have fewer overheads than performing the stride profiling only once. Because unpredictable operations (in terms of using hardware stride value predictors) may have many different stride values, the stride profiler must spend much time searching a profile table and recording new strides into the table. Due to limitations of memory space and table-searching time, performing the stride profiling only once may be inefficient.

In our experiments, integer-register-writing operations in the top 20 treegions [17] in the SPECint95 benchmarks were profiled. For the purpose of comparing the performance of different value predictors, hardware stride and hardware stride two-delta value predictors [10], [13] were experimented. (Note that the difference between hardware stride and hardware stride two-delta value predictors is their stride update policies. The stride of the former always equals the difference between the last two actual results, but the latter updates the stride when the same stride appears at least twice in a row.) Figure 5.3 shows the results of the first profiling using hardware stride and hardware stride two-delta value predictors. The hardware stride two-delta value predictor obtains higher prediction accuracies than the hardware stride value predictor, because the two-delta stride update policy is better by tolerating one value misprediction before changing strides. Using the hardware stride two-delta value predictor, 124.m88ksim has the highest prediction accuracy of 88.83%, and is a good candidate for value speculation. 134.perl is the second with 78.02%. 147.vortex has the third highest value prediction accuracy of 67.42%. 126.gcc, 129.compress, 130.li and 132.ijpeg have moderate value prediction accuracies between 48.38% and 61.21%. 099.go has the lowest value prediction accuracy of 34.37%, and may not be suitable for value speculation. The average prediction accuracy using the hardware stride two-delta value predictor is 61.01%, which is higher than 54.78% by using the hardware stride value predictor.

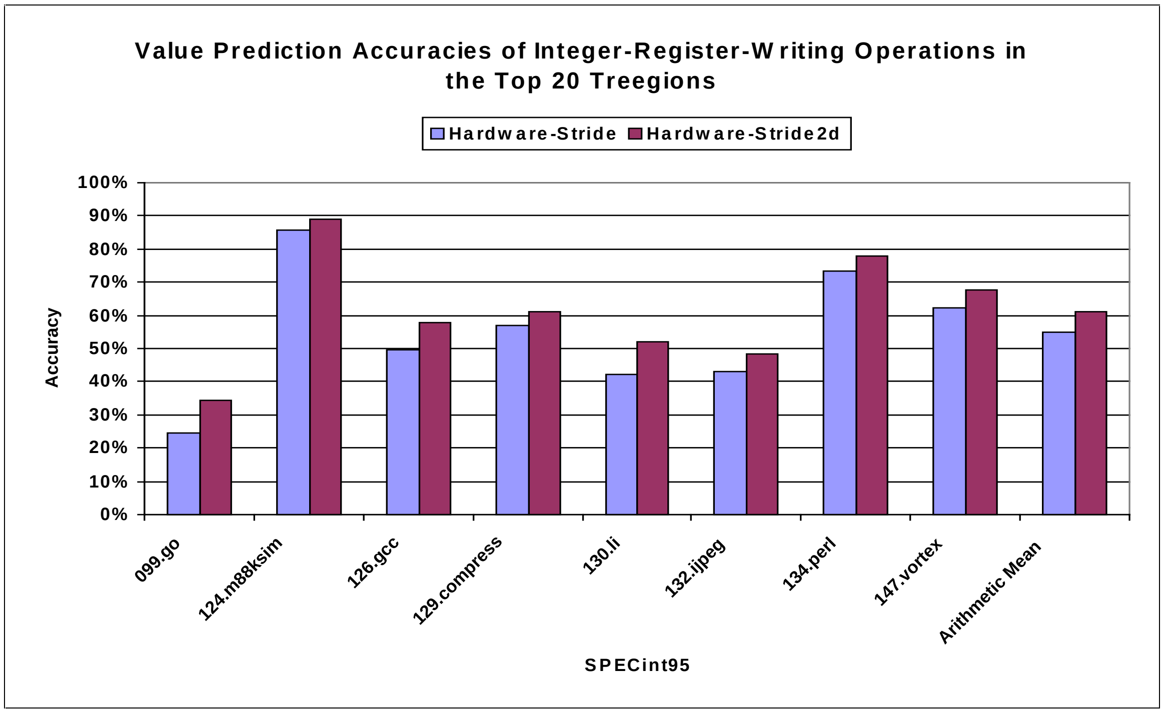


Figure 5.3 Value prediction accuracies of integer-register-writing operations in the top 20 treegions in SPECint95 using hardware stride and hardware stride two-delta value predictors.

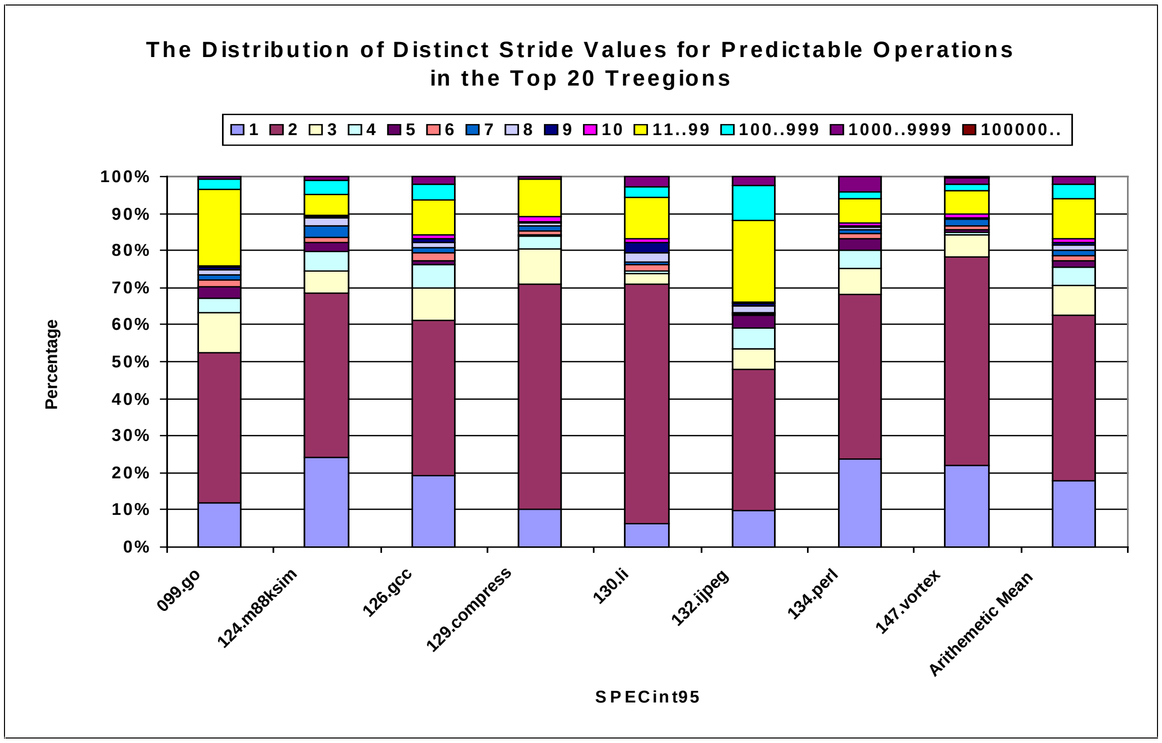


Figure 5.4 The distribution of distinct stride values for predictable operations in the top 20 treegions in SPECint95.

Table 5.1 The top five stride values for predictable operations in SPECint95. In each grid, the first number is a stride value and the second number in parentheses is its corresponding percentage.

Rank	1	2	3	4	5
099.go	0 (97.46%)	4 (0.90%)	1 (0.78%)	-4 (0.39%)	2 (0.16%)
124.m88ksim	0 (85.79%)	1 (4.55%)	12 (2.75%)	4 (2.54%)	24 (1.27%)
126.gcc	0 (95.50%)	1 (1.72%)	4 (1.68%)	-4 (0.23%)	16128 (0.21%)
129.compress	0 (73.78%)	1 (10.27%)	-64 (4.59%)	-1 (2.43%)	8 (2.16%)
130.li	0 (83.09%)	-60 (4.95%)	1 (2.62%)	-1 (1.74%)	12 (1.74%)
132.ijpeg	0 (71.17%)	4 (7.22%)	1 (4.80%)	2 (3.07%)	32 (2.45%)
134.perl	0 (85.31%)	1 (4.03%)	4 (2.66%)	64 (1.90%)	-1 (1.29%)
147.vortex	0 (96.34%)	4 (1.71%)	1 (1.44%)	-4 (0.16%)	8 (0.16%)

After the value prediction accuracy profiling, the stride profiling was performed. Figure 5.4 shows the distribution of distinct stride values for predictable operations in the top 20 treegions. 60% of predictable operations have only one or two distinct stride values. About 20% of predictable operations have more than 10 stride values. The results in Figure 5.4 can show that the running time of the stride profiling is not high, because most predictable operations have few distinct stride values. Table 5.1 shows the top five stride values and their corresponding percentages in parentheses. In all SPECint95 benchmarks, zero is the most frequent stride value, and it accounts for between 71.17% and 97.46% of all stride values. This means that most of the predictable operations generate the same value as the last result. Other stride values are 1, -1, 4, and -4 appearing from the second to the fifth rank. Examining the source code, these stride values occur when operations increase or decrease induction variables or pointers (address registers) by 1 (equal to the size of *char*) or 4 (equal to the size of *int*). It is interesting to note that most of the stride values are even numbers, and many of them are the power of two, because the computer uses a binary system internally.

To fully obtain the performance of software static stride value predictors, one important thing should be noted: the register for storing predicted values needs to be preserved across procedures. This could be done by using load and store operations to fill the register from the memory upon the procedure entrance and spill the register to the memory upon the procedure exit. However, the filling and spilling code introduces overhead that reduces the benefits from SVSS [23]. In this chapter, we propose using global registers [32] to preserve predicted values to eliminate the overhead. Modern microprocessors, MIPS R10000 [29], Alpha 21264 [30], and Intel Itanium [27], [28], [31], [41] have 64, 80, 128 physical integer (or general) registers. In MIPS R10000 and Alpha 21264, many registers are used for hardware register renaming or treated as local registers. In Intel Itanium, there are 32 global (static) registers where 18 registers (r14-r31) are scratch registers and may be utilized experimentally for the SVSS optimization. From the experimental results in Section 5.3, up to 30 global registers were required for implementing software static stride value predictors on the 20 most heavily executed paths in each SPECint95 benchmark.

In the experiments, software static stride value predictors using global registers and local registers were simulated. When using local registers, registers for implementing software static stride value predictors are not preserved across procedures. Figure 5.5 shows the value prediction accuracies using hardware stride, hardware stride two-delta, software static stride (global), and software static stride (local) value predictors for the same set of predictable operations in the SPECint95 benchmarks. The average prediction accuracy using the software static stride (global) value predictor is 95.47%, which is higher than 93.87% by using the hardware stride value predictor. The hardware stride two-delta value predictor has the highest value prediction accuracy of 95.81%, slightly better than the software static stride (global) value predictor. The software static stride (local) value predictor suffers from losing register values across procedures and has the lowest value prediction accuracy of 8.68% on an average. When using the software static stride (local) value predictor, 124.m88ksim has the highest prediction accuracy of 34.20%, because many operations are inside *intra-procedural* loops and can be predicted correctly without preserving registers across procedures (*inter-procedurally*).

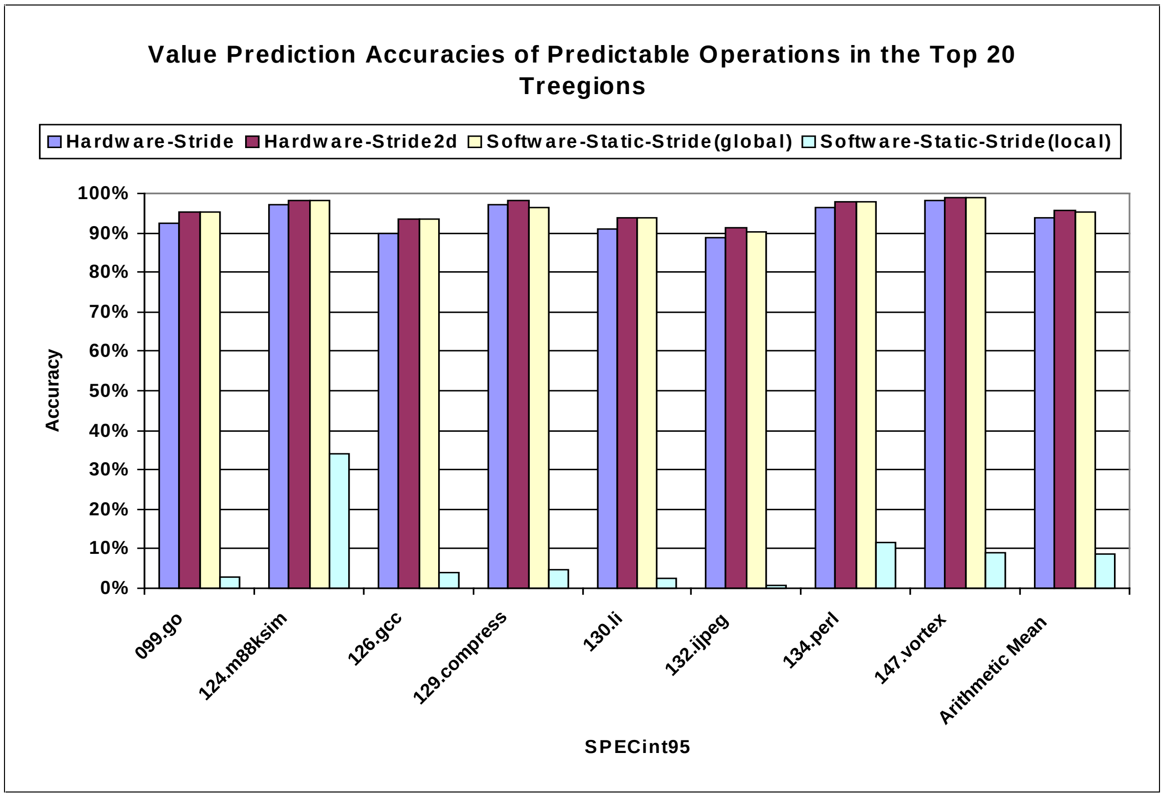


Figure 5.5 Value prediction accuracies using hardware stride, hardware stride two-delta, software static stride (global), and software static stride (local) value predictors for predictable operations in the top 20 treegions in SPECint95.

5.3 Experimental Results

For analyzing benefits, the optimal edge selection algorithm presented in Chapters 3 and 4 was performed on the 20 most heavily executed paths selected from each SPECint95 benchmark. Figure 5.6 shows the speedup on the 20 most heavily executed paths using hardware stride, hardware stride two-delta, and software static stride value predictors. The speedup is calculated as the maximal benefit divided by the critical path length of the original data dependence graph. In Figure 5.6, using hardware stride two-delta value predictors and using software static stride value predictors obtain the same speedup of 9.43% on an average, which is higher than that by using hardware stride value predictors. When using software static stride value predictors, 124.m88ksim has the highest speedup of 24.44%, because 124.m88ksim has the highest value prediction accuracy as shown in Figure 5.3. 147.vortex has the second highest speedup of 17.50%. 099.go and 132.ijpeg have zero or very small speedups, because of their low value prediction accuracies. The other benchmarks show significant speedups between 6% and 11% that are available for SVSS to exploit.

After running the benefit analysis, both maximal benefits and optimal sets of edges for SPECint95 benchmarks were found. All SPECint95 benchmarks except 099.go and 132.ijpeg were chosen for performing SVSS on the 20 most heavily executed paths. The SPECint95 programs were compiled with classic optimizations by the IMPACT compiler from the University of Illinois [18] and converted to the Rebel textual intermediate representation by the Elcor compiler from Hewlett-Packard Laboratories [19]. Then, the LEGO compiler [17] scheduled base code and SVSS-optimized code on a 16-issue VLIW machine model based on the Hewlett-Packard Laboratories HPL-PD architecture [20]. All operations have a one-cycle latency except for load (two cycles), floating-point add (two cycles), floating-point subtract (two cycles), floating-point multiply (threes cycles) and floating-point divide (three cycles).

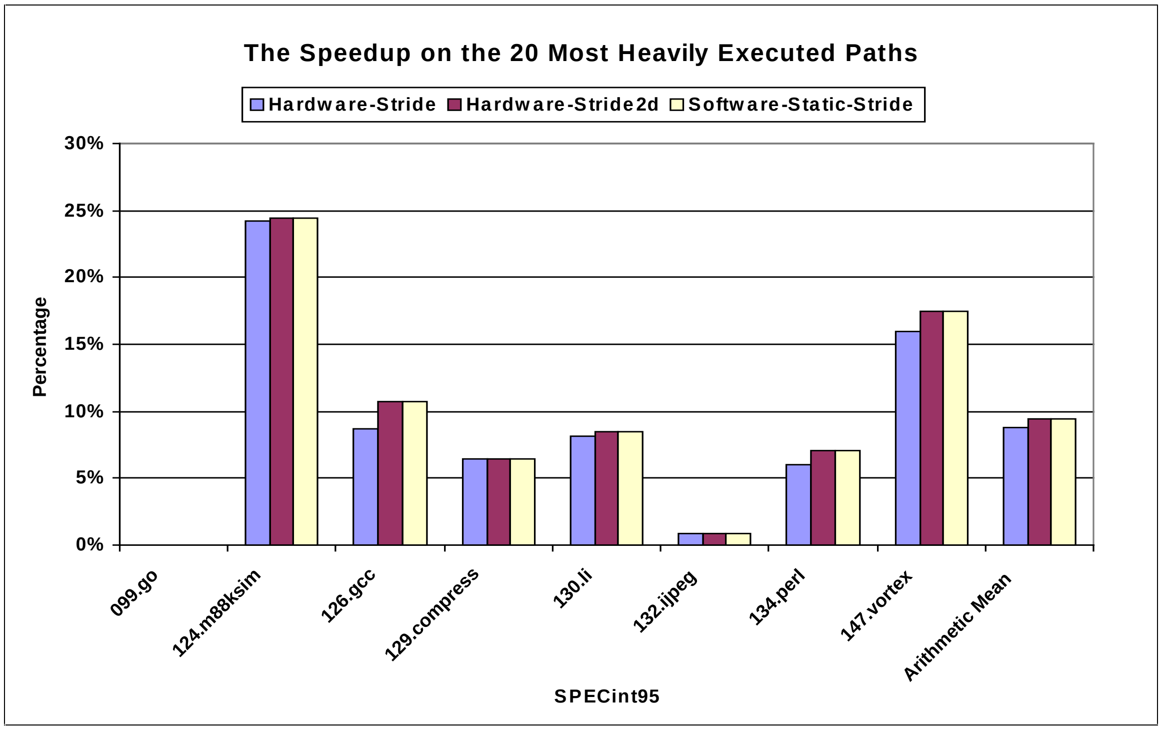


Figure 5.6 The speedup on the 20 most heavily executed paths in each SPECint95 benchmark using hardware stride, hardware stride two-delta, and software static stride value predictors. (Note that 099.go has no speedups in all cases.)

As described in Section 5.2, global registers are used for preserving predicted values across procedures. Table 5.2 shows the number of global registers required for implementing software static stride value predictors on the 20 most heavily executed paths. 124.m88ksim requires 30 global registers, which is the most. Other benchmarks need between 6 and 11 global registers. In the experiments, in additional to the 128 integer registers, up to 30 global registers were used for the compiler to implement software static stride value predictors.

Table 5.2 The number of global registers required for implementing software static stride value predictors on the 20 most heavily executed paths in SPECint95.

SEPCint95	124.m88ksim	126.gcc	129.compress	130.li	134.perl	147.vortex
# of Global Registers	30	9	6	6	11	11

Table 5.3 Three 16-issue VLIW machine models.

Machine	Configuration
Model 1	Dispatch/issue/retire bandwidth: 16 Universal functional units: 16 I-Cache: ideal D-Cache: ideal Branch predictor: ideal
Model 2	Dispatch/issue/retire bandwidth: 16 Universal functional units: 16 I-Cache: Compressed (zero-nop) and two banks with 64k bytes [33] Line size = 16 operations (each bank) Miss penalty = 12 cycles D-Cache: Size/assoc./repl. = 64kB/4-way/LRU Line size = 32 bytes Miss penalty = 14 cycles Branch predictor: multi-way branch prediction [34], [35] Branch prediction table (BPT) = 2^{14} entries Branch target buffer (BTB) entry/assoc./repl. = 2^{14}/8-way/LRU Branch misprediction stalls = 5 cycles
Model 3	Same as Model 2 except branch misprediction stalls = 10 cycles

Trace simulation was performed for three different 16-issue VLIW machine models shown in Table 5.3. In Table 5.3, Model 1 represents an ideal machine model without I-Cache stalls, D-Cache stalls, and branch misprediction penalties. Model 2 represents a realistic model with a 64k-byte compressed I-Cache [33], a 64k-byte D-

Cache, and a multi-way branch predictor [34], [35] that has five-cycle stalls, 2^{14} entries in the branch prediction table (BPT), and 2^{14} entries in the branch target buffer (BTB). Model 3 has a similar configuration to Model 2, but its branch misprediction stalls are 10 cycles. Figure 5.7 shows the execution time speedup of SVSS-optimized code over base code (without applying SVSS) on three machine models. By using Model 1, all SVSS-optimized programs have positive speedups that represent ideal program execution in the processor pipeline. 147.vortex gets the highest speedup of 14.58%, and 124.m88ksim is second with 10.37%. 129.compress and 130.li have moderate speedups, around 2.5%. However, 126.gcc and 134.perl have small speedups of 1%. When using Model 2 and Model 3, all benchmarks except 147.vortex suffer from the side effect of speculative execution on the I-Cache, D-Cache, and multi-way branch predictor. The increased stalls from the I-Cache, D-Cache, and branch predictor decrease the speedups of SVSS-optimized code over base code from 5.25% (Model 1) to 4.10% (Model 2) and 3.67% (Model 3) on a harmonic average. However, 147.vortex is the exception, which yields better speedups on Model 2 and Model 3 than on Model 1. The reason is that SVSS-optimized code has fewer branch misprediction stalls than base code does. After applying SVSS, the control flow in 147.vortex is changed, so that the multi-way branch predictor can predict BNE branches that verify predicted values and other branches more accurately. Using Model 2 and Model 3 enlarges the execution time difference between SVSS-optimized code and base code of 147.vortex. 126.gcc and 134.perl have negative speedups on Model 2 and Model 3. Because their ideal speedups are small on Model 1, the increased I-Cache, D-Cache, branch misprediction stalls counteract the benefits from the SVSS optimization.

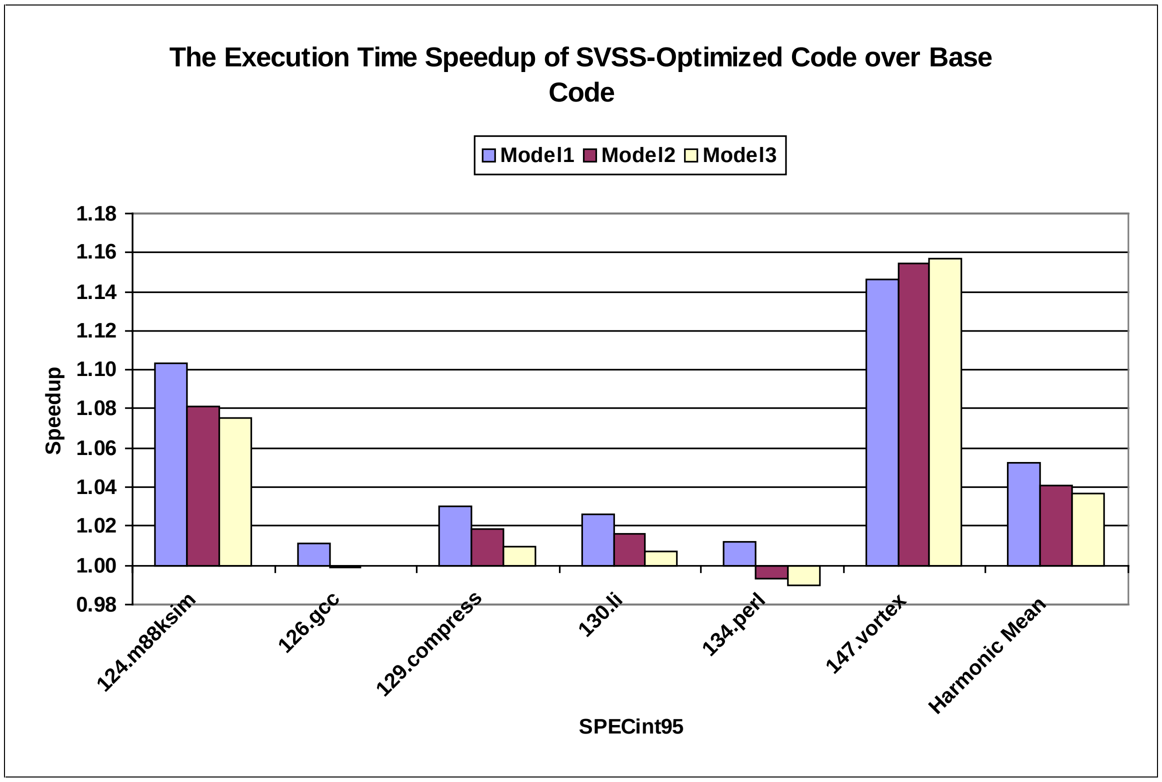


Figure 5.7 The execution time speedup of SVSS-optimized code using software static stride value predictors over base code. (Note that the speedups of 126.gcc are slightly less than 1.00 on Models 2 and 3.)

5.4 Summary

In this chapter, we propose software-only value speculation scheduling (SVSS) to improve the performance of microprocessors by utilizing software static stride value predictors. SVSS has the advantage of being applicable to existing microprocessors without adding new hardware value predictors and modifying processor pipelines. From the experimental results, the prediction accuracy using the software static stride value predictor is comparable to that using the hardware stride two-delta value predictor. The benefit analysis shows that by using the software static stride value predictor, averaging 9.43% of critical path reduction can be obtained on the 20 most heavily executed paths in each SPECint95 benchmark. The overall execution time speedup of SVSS-optimized code over base code on a 16-issue VLIW machine model is encouraging with up to 15%, and averaging 4%.

Future work will include the design of new software value predictors to predict operations with different patterns other than strides. The SVSS scheme can be experimented in Intel Itanium [27], [28], [31], [41] to evaluate the effectiveness of value speculation. For architectures that have very few global registers available for the compiler usage, a new register file may be created. Special-purpose ADD and MOVE operations can access the new register file to implement software static stride value predictors for the SVSS optimization.

Chapter 6

Hardware-Based Value Profiling

Program profiling [4], [7], [39] is a mechanism to collect information about a program. The profile results may include execution frequencies of basic blocks, miss rates of the I-Cache or D-Cache, prediction accuracies of branches or operations, and the distribution of executed operations. Several applications [22], [23], [39] utilize different kind of profile information. For the VSS optimization [22], the compiler relies on the predictability of operations to make judicious decisions of selecting and breaking flow dependences. To obtain the prediction accuracies of operations, the program is instrumented with additional code that simulates hardware value predictors. Then, running the instrumented program with training inputs generates the value prediction accuracies of profiled operations. The process of program profiling has the disadvantage of large overheads when running the instrumented program. Also, the training inputs for program profiling must be the representative to other runs, such that the profile information may be useful for program optimization.

Compared to program profiling, hardware-based profiling [37], [40] is a better technique to gather information at run-time with fewer overheads and more accuracies to actual usage of a program. In this chapter, we adopt the same concept of hardware-based profiling [37], [40] and propose hardware-based value profiling to recognize highly predictable operations at run-time. An augmented value predictor that has between 16 and 256 entries is experimented to profile operations at the retirement stage. Each entry in the value predictor contains a tag and a saturating profile counter. The tag stores the instruction pointer (PC) or the compiler-assigned index of the operation. The profile counter indicates the predictability of operations. Upon context-switches or interrupts, the tags with the maximum saturating counter values are stored to memory, so only highly predictable operations are recorded. From the experimental results, hardware-based value profiling can accurately identify highly predictable operations. Using the value predictor with 256 entries gathers 25% of static profiled operations that account for 43% of dynamic profiled operations. The collected operations have very high prediction accuracies of 94% that can be utilized by the VSS optimization.

The remainder of this chapter is organized as follows. Section 6.1 measures the profile shifts under different input sets to investigate if the profile information remains invariant. The profile invariance is important to all profile-driven optimizations. Section 6.2 proposes a scheme of hardware-based value profiling. Section 6.3 presents experimental results of hardware-based value profiling. Section 6.4 concludes this chapter.

6.1 Profile Invariance

The profile results are useful for program optimization only if the collected information remains invariant among different runs. The invariant characteristics are important to design hardware-based value profiling as well. In this section, the predictability of operations is profiled using different inputs to measure the profile shift. The results of the profile shift will guide us to design a scheme of hardware-based value profiling.

In the experiments, the SPECint95 benchmark suite was used with three different input sets, *train*, *test*, and *ref*, which are shown in Tables 6.1, 6.2, and 6.3. All integer-register-writing operations in each SPECint95 program were profiled using a hybrid value predictor [13], [22]. The hybrid value predictor [13], [22] contains stride [4], [10], [13] and context-based value predictors [10], [11]. The value prediction table size equals the number of all integer-register-writing operations in each SPECint95 program. For the stride value predictor, each entry records the last actual result and the stride of the last two values. Adding the last value and the stride generates a prediction. For the context-based value predictor, the entry in the first level table records one actual result that indexes a local second level table with 16 entries to generate a prediction. The hybrid predictor selects a prediction between the stride value predictor and the context-based value predictor based on counters associated with each value predictor. When the value predictor generates a correct prediction, the counter increases by three, and up to twelve. In the case of value misprediction, the counter decreases by one, and down to zero.

Table 6.1 The train input set for the SPECint95 benchmarks.

SPECint95	Train Inputs
099.go	go 50 9 2stone9.in
124.m88ksim	m88ksim -c < ctl.raw
126.gcc	gcc -quiet -funroll-loops -fforce-mem -fcse-follow-jumps -fcse-skip-blocks -fexpensive-optimizations -fstrength-reduce -fpeephole -fschedule-insns -finline-functions -fschedule-insns2 -O amptjp.i -o amptjp.s
129.compress	compress95 < test.in
130.li	li train.lsp
132.ijpeg	ijpeg -image_file vigo.ppm -compression.quality 90 -compression.optimize_coding 0 -compression.smoothing_factor 90 -difference.image 1 -difference.x_stride 10 -difference.y_stride 10 -verbose 1 -GO.findoptcomp > vigo.out
134.perl	perl jumble.pl < jumble.in
147.vortex	vortex vortex.raw

Table 6.2 The test input set for the SPECint95 benchmarks.

SPECint95	Test Inputs
099.go	go 40 19 null.in
124.m88ksim	m88ksim -c < ctl.raw
126.gcc	gcc -quiet -funroll-loops -fforce-mem -fcse-follow-jumps -fcse-skip-blocks -fexpensive-optimizations -fstrength-reduce -fpeephole -fschedule-insns -finline-functions -fschedule-insns2 -O cccp.i -o cccp.s
129.compress	compress95 < test.in
130.li	li test.lsp
132.ijpeg	ijpeg -image_file specmun.ppm -compression.quality 90 -compression.optimize_coding 0 -compression.smoothing_factor 90 -difference.image 1 -difference.x_stride 10 -difference.y_stride 10 -verbose 1 -GO.findoptcomp > specmun.out
134.perl	perl primes.pl < primes.in
147.vortex	vortex vortex.raw

Table 6.3 The ref input set for the SPECint95 benchmarks.

SPECint95	Ref Inputs
099.go	go 50 21 9stone21.in
124.m88ksim	m88ksim -c < ctl.raw
126.gcc	gcc -quiet -funroll-loops -fforce-mem -fcse-follow-jumps -fcse-skip-blocks -fexpensive-optimizations -fstrength-reduce -fpeephole -fschedule-insns -finline-functions -fschedule-insns2 -O 2stmt.i -o 2stmt.s
129.compress	compress95 < bigtest.in
130.li	li *.lsp
132.ijpeg	ijpeg -image_file penguin.ppm -compression.quality 90 -compression.optimize_coding 0 -compression.smoothing_factor 90 -difference.image 1 -difference.x_stride 10 -difference.y_stride 10 -verbose 1 -GO.findoptcomp > penguin.out
134.perl	perl primes.pl < primes.in
147.vortex	vortex vortex.raw

Table 6.4 Statistics of total static and dynamic operations in the SPECint95 benchmarks using train, test and ref input sets.

Input Sets	Train		Test		Ref	
#	Static	Dynamic	Static	Dynamic	Static	Dynamic
099.go	28,189	297,736,250	33,056	9,122,826,104	32,723	18,617,385,077
124.m88ksim	3,682	72,999,794	4,362	274,921,513	5,537	41,943,677,738
126.gcc	76,174	625,895,979	75,280	616,745,988	70,883	272,487,688
129.compress	543	25,210,466	485	2,778,460	632	29,861,979,006
130.li	1,677	106,801,597	1,584	588,167,531	2,342	34,095,697,502
132.ijpeg	6,866	1,193,881,971	6,771	436,352,849	6,798	24,697,603,777
134.perl	7,098	939,198,744	5,219	4,503,008	5,225	8,089,016,397
147.vortex	33,067	1,414,769,366	33,172	5,116,326,427	33,132	43,854,292,815
Average	**19,662**	**584,561,771**	**19,991**	**2,020,327,735**	**19,659**	**25,179,017,500**

After program profiling, the statistics of operations using train, test, and ref input sets are shown in Table 6.4. Each input set contains two columns, *static* and *dynamic*.

The number in the static column indicates the number of static operations that are executed, and the number of static operations multiplying the execution frequency generates the number in the dynamic column. For all three input sets, the average numbers of static operations are close, 19,662 (train), 19,991 (test), and 19,659 (ref). However, the average numbers of dynamic operations are very different in three input sets. The train input set has the fewest dynamic operations of 584 million. The test input set contains 2 billion dynamic operations, and the ref input set has the most dynamic operations of 25 billion.

Based on the profile results of value prediction accuracies by running the train input set, the profile shift against the test input set is presented in Figures 6.1 and 6.2, and the profile shift against the ref input set is presented in Figures 6.3, and 6.4. In Figure 6.1, the value prediction accuracies of all profiled operations that appear both in the train and test input sets are compared. In Figure 6.2, only operations that have prediction accuracies higher than 90% in the train input set are further considered to be compared with operations in the test input set. The profile shift is calculated as the value prediction accuracy difference of operations between two different input sets. Figures 6.1 and 6.2 show the distribution of the value prediction accuracy differences between the train and test input sets. In Figure 6.1, averaging 80% of all profiled operations have value prediction accuracy differences less than 10%. In Figure 6.2, averaging 95% of predictable operations have value prediction accuracy differences less than 10%. This shows that under different input sets, highly predictable operations are more invariant than all operations are.

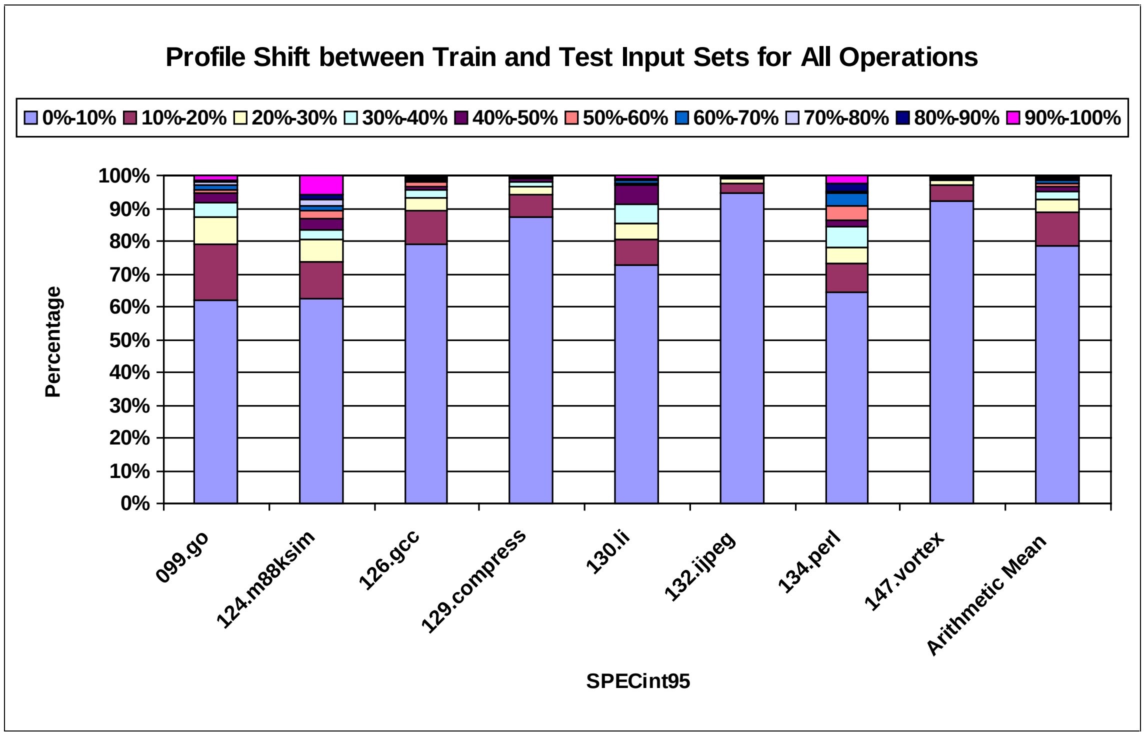


Figure 6.1 Profile shift between train and test input sets for all integer operations.

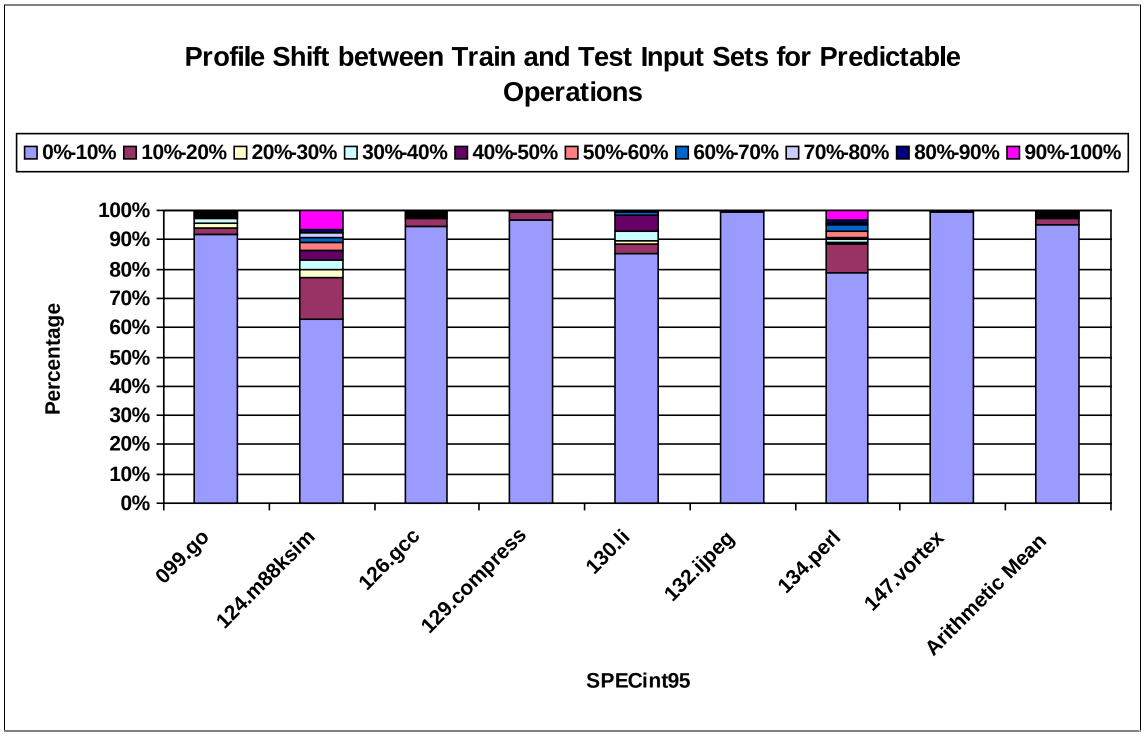


Figure 6.2 Profile shift between train and test input sets for predictable integer operations whose prediction accuracies are higher than 90%.

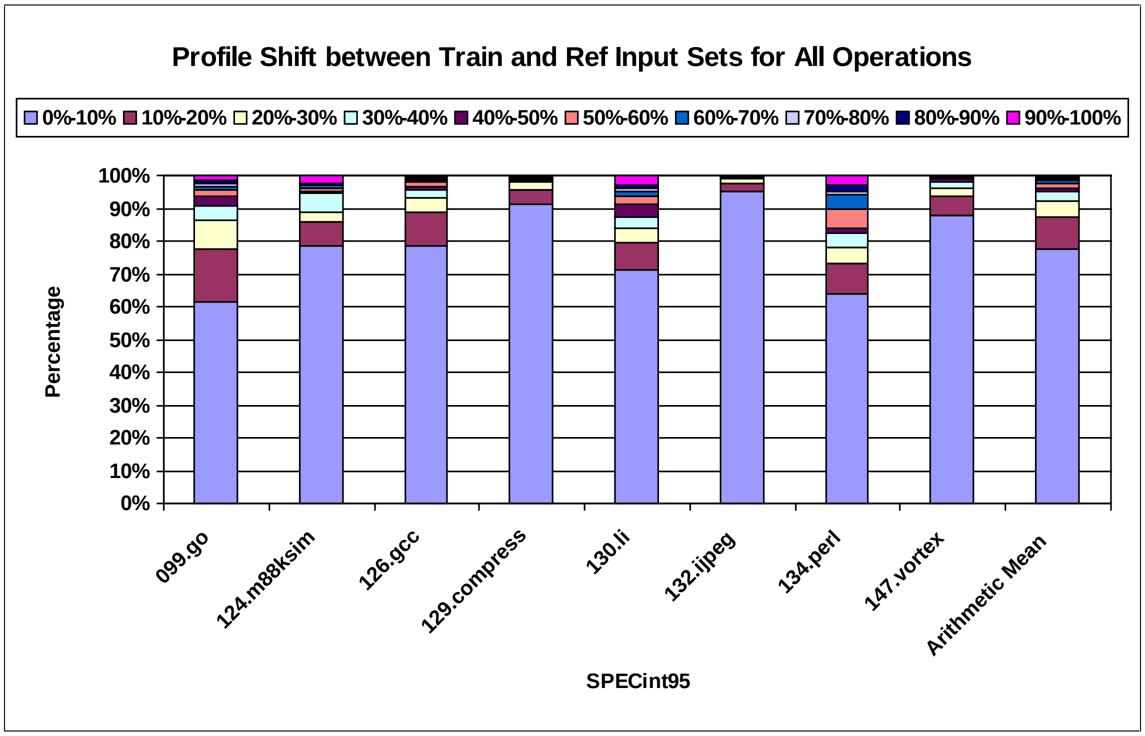


Figure 6.3 Profile shift between train and ref input sets for all integer operations.

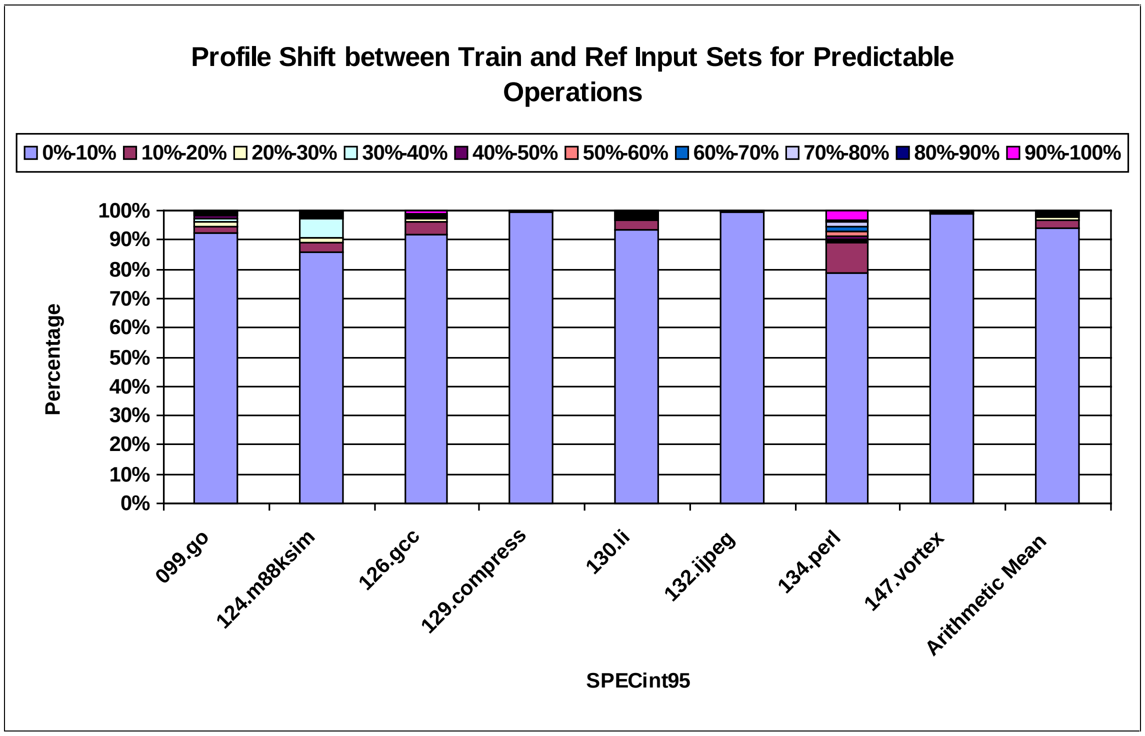


Figure 6.4 Profile shift between train and ref input sets for predictable integer operations whose prediction accuracies are higher than 90%.

Comparing the benchmarks in Figure 6.2, 132.ijpeg and 147.vortex are the most invariant programs that have almost 100% of predictable operations with value prediction accuracy differences less than 10%. However, 124.m88ksim and 134.perl have 40% and 20% of predictable operations that experience more than 10% of value prediction accuracy differences between the train and test input sets. The operations with varied predictability may hurt the program performance, if they are selected for the VSS optimization.

Figures 6.3 and 6.4 show the distribution of the value prediction accuracy differences between the train and ref input sets. Figure 6.3 has the similar trend to Figure 6.1. Averaging 80% of static operations have value prediction accuracy differences less than 10%. However, Figure 6.4 shows better profile invariance than Figure 6.2 does. Especially for 124.m88ksim, more than 85% of predictable operations have value prediction accuracy differences less than 10%. It means that in 124.m88ksim, the executed operations are strongly correlated between the train and ref input sets. From the results of Figure 6.4, if the ref input set is the input that the user runs, the profile information from running the train input set will be good feedbacks to perform the VSS optimization.

From Figures 6.2 and 6.4, predictable operations are more invariant than all operations. Also, from the optimization's point of view, the highly predictable operations are candidates for the VSS optimization. Therefore, a scheme of hardware-based value profiling is designed to identify and collect highly predictable operations.

6.2 Hardware-Based Value Profiling

Based on the profile invariance of predictable operations in Section 6.2, a scheme of hardware-based value profiling is proposed in Figure 6.5 to collect highly predictable operations. The process of hardware-based value profiling occurs at the retirement stage, such that it is not on the critical path of the processor pipeline. In Figure 6.5, the scheme of hardware-based value profiling can be applied to different prediction mechanisms, e.g., last value prediction [2], [3], stride value prediction [4], [10], [13], context-based value prediction [10], [11], or hybrid value prediction [13], [22].

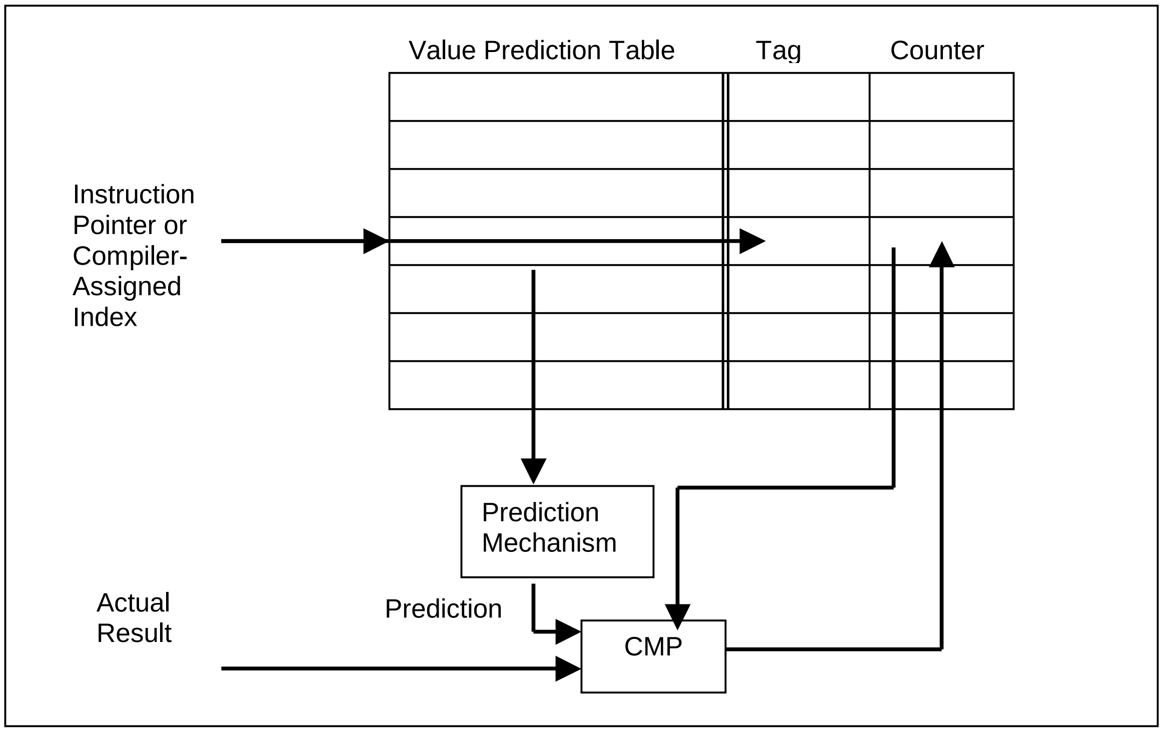


Figure 6.5 A scheme of hardware-based value profiling.

In Figure 6.5, each entry of the value prediction table has two new fields: a *tag* and a saturating profile *counter*. The tag records an instruction pointer (PC) or a compiler-assigned index. The counter indicates the predictability of operations. The processes of hardware-based value profiling are as follows. When an integer operation retires, the actual result of the operation is available to be profiled. First, the instruction pointer or the compiler-assigned index selects one entry in the value predictor. Second, the tag from the selected entry is compared with the current index. If they are different, the counter value is read and checked against a threshold. If the current counter value is greater than the threshold, the profiling process stops and nothing happens. The threshold is used to favor predictable operations to occupy the selection entries longer. Third, if the profiling process continues and the old tag is different, the current index is stored to the tag field and the counter value is reset to zero. Fourth, the value predictor generates a prediction based on its prediction mechanism. Fifth, the comparison between the prediction and the actual result is used to update the saturating profile counter. If the prediction is correct, the counter value increases by an amount, up to a maximal value. Otherwise, the counter value decreases by an amount, down to zero. Also, the value predictor is updated by the actual result for future predictions.

Upon context-switches or interrupts, the profile information in the value prediction table can be recorded. To favor highly predictable operations and reduce the overhead of storing data, only the tags with the maximal saturating counter values are stored to the memory. After recording the tags, all tags and counters in the value prediction table are reset to zero for the next profiling run. From [37] and [40], hardware-based branch profiling has a slowdown of 1.02 on an average, and 1.05 at a worst case. Because hardware-based value profiling uses the similar scheme to hardware-based branch profiling, the slowdown is expected to be small.

6.3 Experimental Results

In this section, the scheme of hardware-based value profiling using the hybrid value predictor, which is described in Section 6.1, was experimented. The saturating profile counter increased by 1, up to 15, and decreased by 5, down to 0. The threshold value of the profile counter was 8. As described in Section 6.2, if the current counter value is less than 8, the new profiled operation can continue the profiling process by replacing the tag and resetting the counter to zero. The interrupt occurred every one million profiled operations to store the tag fields to the memory. The value prediction table size was varied with 16, 32, 64, 128, and 256 entries. The compiler selected integer-register-writing operations in the top 20 treegions [17] for hardware-based value profiling, and sequentially assigned the indices to the profiled operations. For comparing the performance, program profiling [4], [7] using the hybrid value predictor was experimented. The results of program profiling serve as the upper bound for the scheme of hardware-based value profiling to compare, because program profiling does not encounter conflicts in the simulated value prediction table.

Table 6.5 shows the number of static operations, the number of dynamic operations, and the prediction accuracies using program profiling. In Table 6.5, 126.gcc has the most static operations of 13,567, and 129.compress has the fewest static operations of 543. The average number of static operations is 3,872. If there are too many static operations to be profiled, the conflicts in the value prediction table will increase under the scheme of hardware-based value profiling. The value prediction accuracies using the hybrid predictor vary in different benchmarks, and the average prediction accuracy is 61.05%.

Table 6.5 Statistics of profiled operations and prediction accuracies using program profiling.

SPECint95	# of Static Operations	# of Dynamic Operations	Prediction Accuracies
099.go	5,805	261,702,531	32.42%
124.m88ksim	1,619	69,550,441	90.18%
126.gcc	13,567	339,029,896	57.58%
129.compress	543	25,210,466	66.54%
130.li	784	102,766,436	50.75%
132.ijpeg	2,394	1,127,763,347	45.14%
134.perl	2,837	896,156,199	79.65%
147.vortex	3,428	1,040,658,220	66.10%
Arithmetic Mean	**3,872**	**482,854,692**	**61.05%**

Table 6.6 Statistics of profiled operations whose value prediction accuracies are higher than 90% using program profiling.

SPECint95	Percentage of static operations whose accuracies > 90%	Percentage of dynamic operations whose accuracies > 90%
099.go	18.35%	14.46%
124.m88ksim	49.41%	80.75%
126.gcc	39.29%	32.99%
129.compress	63.35%	58.11%
130.li	33.16%	26.09%
132.ijpeg	43.94%	22.46%
134.perl	40.85%	68.09%
147.vortex	38.77%	55.99%
Arithmetic Mean	**40.89%**	**44.86%**

The percentages of static and dynamic predictable operations whose prediction accuracies are higher than 90% using program profiling are shown in Table 6.6. These numbers will be used to evaluate the coverage of predictable operations when using hardware-based value profiling. The average percentages of static and dynamic predictable operations are 40.89% and 44.86%. In Table 6.5, 124.m88ksim has the highest value prediction accuracy of 90.18%, so the percentage of dynamic predictable operations is the most among the SPECint95 benchmarks.

Figures 6.6 and 6.7 show the percentages of static and dynamic operations that are selected and stored to the memory by using hardware-based value profiling with 16, 32, 64, 128, and 256 entries in the value predictor. For all benchmarks, the percentage of selected static operations increases when the number of value predictor entries increases, because fewer conflicts occur in the value prediction table. When doubling the table size from 16 to 32, 64, 128, 256 entries, the coverage of static and dynamic predictable operations increases for most of the cases. The diminishing return appears when doubling from 128 to 256 entries. In 126.gcc, because 13,567 operations are selected to be profiled, many conflicts in the value prediction table affect the number of predictable operations that can be recorded by hardware-based value profiling. In Figure 6.6, using the value predictor with 256 entries collects 25% of static operations on an average. As shown in Table 6.6, the average percentage of static operations whose prediction accuracies are higher than 90% is 40%, which is 15% more than the percentage by using hardware-based value profiling with 256 entries. However, in Figure 6.7, using hardware-based value profiling with 256 entries covers 43% of dynamic operations, which are very similar to the percentage of dynamic operation whose accuracies are higher than 90% by using program profiling that is shown in Table 6.6.

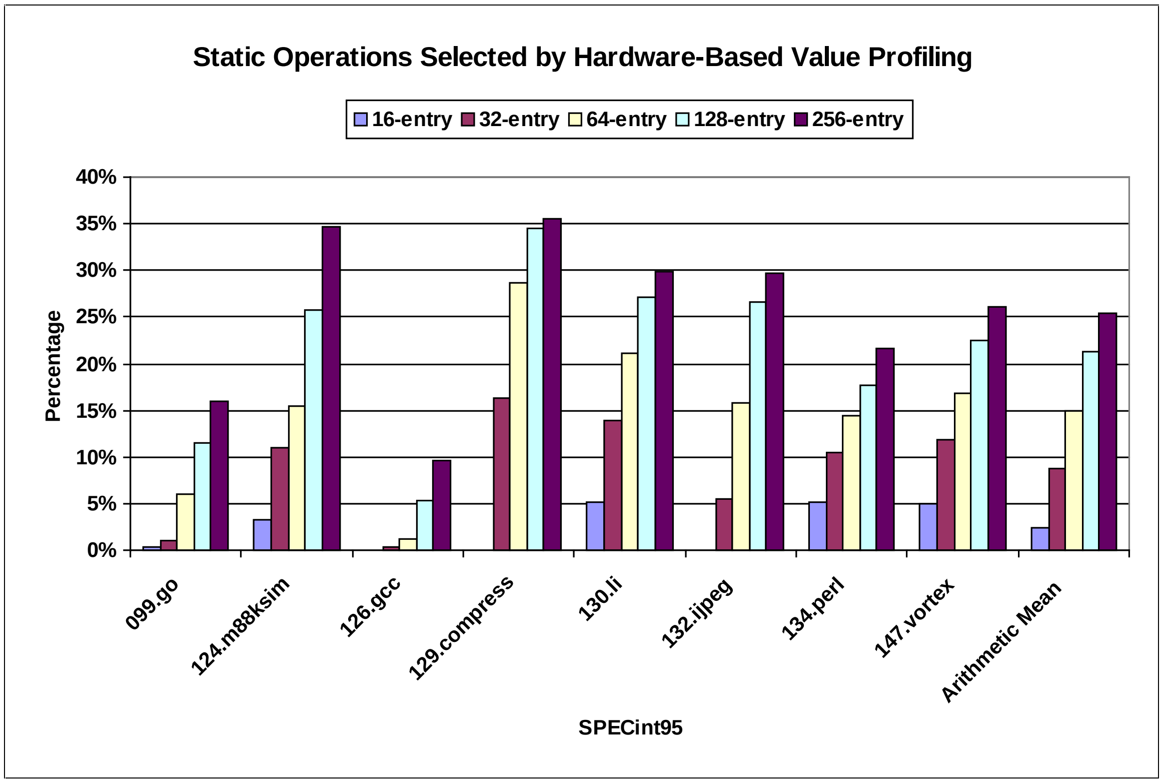


Figure 6.6 Statistics of static operations selected by hardware-based value profiling.

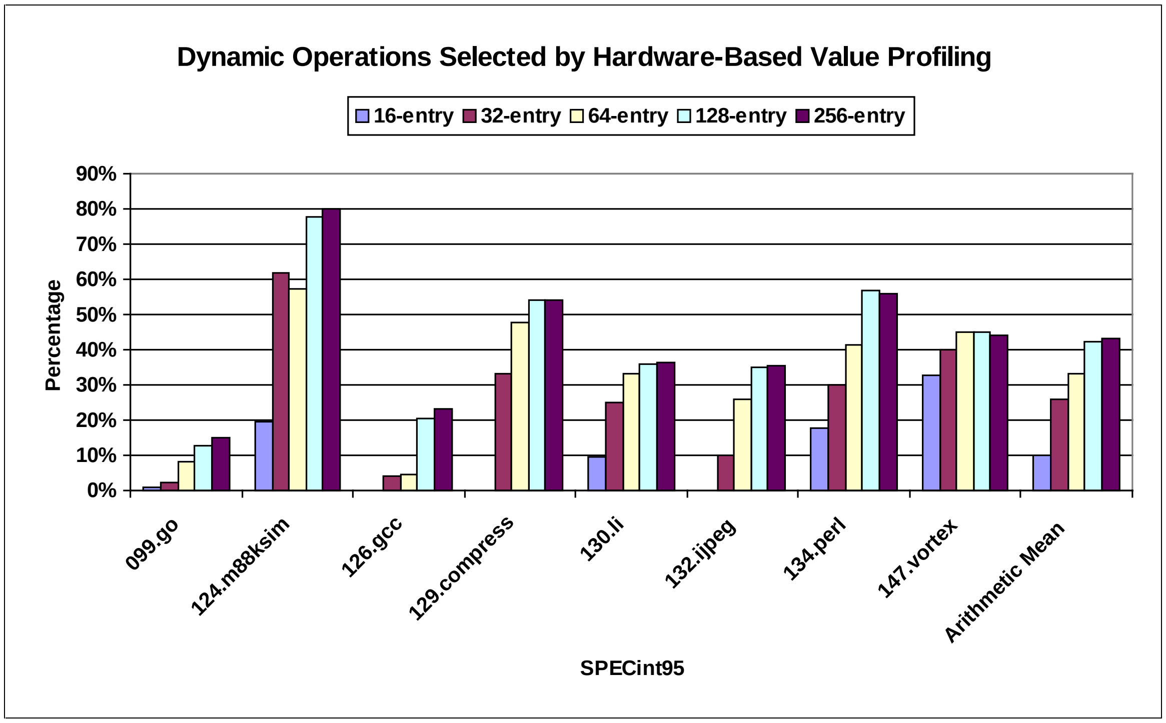


Figure 6.7 Statistics of dynamic operations selected by hardware-based value profiling.

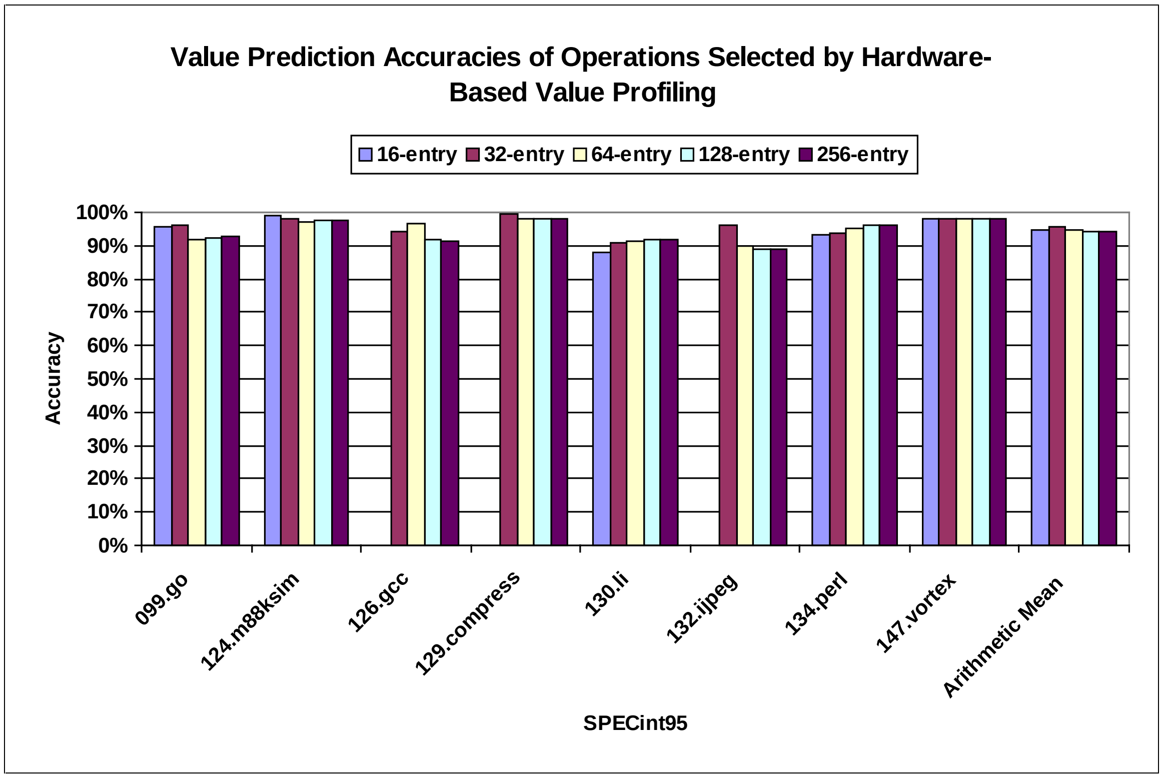


Figure 6.8 Value prediction accuracies of operations selected by hardware-based value profiling. Some value prediction accuracies are zeros, because no operations are collected under these schemes.

Figure 6.8 shows the value prediction accuracies of operations that are collected by hardware-based value profiling with 16, 32, 64, 128, and 256 entries. Some value prediction accuracies are zero in Figure 6.8, because no operations are collected under these schemes. Using different sizes of the value prediction table, the average value prediction accuracies are around or above 90%. Profiled operations in 124.m88ksim, 129.compress, and 147.vortex have very high prediction accuracies of 98%. The highly predictable operations can be utilized by the VSS optimization. For all benchmarks, by using different value prediction table sizes, the average prediction accuracies are around 94%. Figure 6.8 can show that hardware-based value profiling accurately collects highly predictable operations.

One application of hardware-based value profiling is to feed the collected highly predictable operations to perform the VSS optimization. As described in Chapter 4, solving an optimal edge selection problem in a data dependence graph serves as a compilation phase of the benefit analysis. For the benefit analysis, the value misprediction rates and branch misprediction rates are required to model the penalties for mispredicting operations (from Figure 3.5). However, hardware-based value profiling only records the indices of profiled operations that have maximal saturating counter values upon context-switches or interrupts. As shown in Figure 6.9, the value misprediction rates and branch misprediction rates need to be synthesized for the scheme of hardware-based value profiling. In Figure 6.9, the number of occurrences for each index is used to calculate the value misprediction rate. The branch misprediction rate is set to be the same as the value misprediction rate, because highly value-predictable operations have corresponding highly predictable BNE branches (from Figure 3.14).

```
Value_misprediction_rate = 1 /  (15 ^ number_of_occurrences)

Branch_misprediction_rate = Value_misprediction_rate
```

Figure 6.9 The synthesized value misprediction rates and branch misprediction rates for the scheme of hardware-based value profiling.

Based on the synthesized value misprediction rates and branch misprediction rates for the scheme of hardware-based value profiling with 256 entries, the VSS optimization was performed on all SPECint95 benchmarks except for 099.go and 132.ijpeg. For comparison, the VSS optimization was also performed based on the results of program profiling. Figure 6.10 shows the execution time speedup of VSS-optimized code over base code using the feedbacks from program profiling and hardware-based value profiling. The machine model is the same as Model 2 in Table 5.3 with the I-Cache, D-Cache, and multi-way branch predictor with 5-cycle stalls.

In Figure 6.10, for all benchmarks except 147.vortex, using the feedbacks from program profiling to perform VSS obtains larger speedups than using the synthesized data from hardware-based value profiling. The differences of speedups are significant. The reason is that program profiling gathers precise value prediction accuracies of operations, so that the benefit analysis can accurately find an optimal set of dependences to be broken via VSS. However, 147.vortex, which contains many very highly predictable operations, is the exception. Because the synthesized data for 147.vortex indicates that many operations are highly predictable, the VSS optimization is performed more aggressively under the scheme of hardware-based value profiling than under program profiling, and the resultant speedup is higher. In 126.gcc and 134.perl, the speedups are negative when using the feedbacks from hardware-based value profiling, because too many candidates are selected for the VSS optimization and resultant penalties for value-speculative execution increase.

For all benchmarks, the harmonic average speedup using program profiling is 5.5% that is higher than 3.5% using hardware-based value profiling. In general, using the feedbacks from hardware-based value profiling enables more candidates for the VSS optimization. The aggressive VSS optimization hurts the performance for most of the benchmarks. In future work, for the scheme of hardware-based value profiling, the synthesized value misprediction rates and branch misprediction rates can be adjusted to be higher for performing the VSS optimization conservatively.

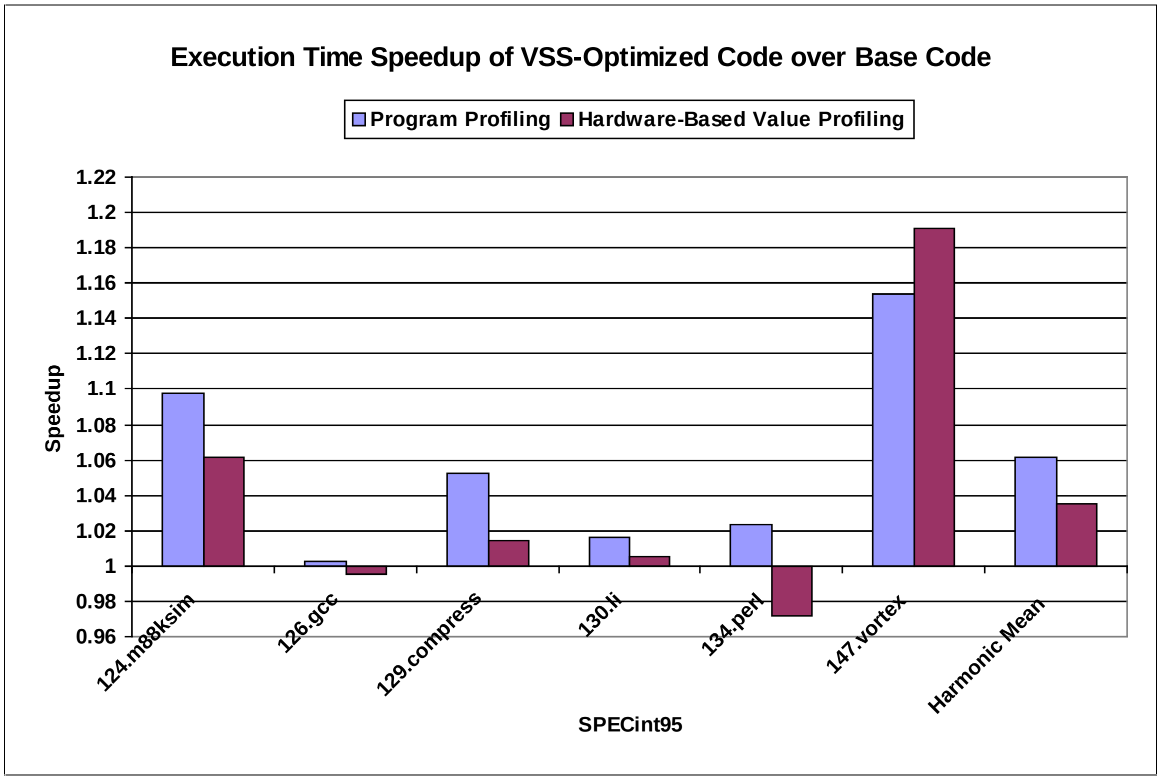


Figure 6.10 The execution time speedup of VSS-optimized code over base code using the feedbacks from program profiling and hardware-based value profiling.

6.4 Summary

In this chapter, hardware-based value profiling is proposed to reduce the overhead of program profiling and eliminate the need of profile training inputs. From the experimental results of running different input sets for the SPECint95 benchmarks, the highly predictable operations are invariant and need to be collected by hardware-based value profiling. The value predictor with additional tag and counter fields is proposed as the scheme of hardware-based value profiling. At the retirement stage, operations access the value predictor and update the tag and counter fields. Upon context-switches or interrupts, the tags with the maximal saturating counter values are stored to the memory for recording predictable operations. In the experiments, using the value predictor with 16, 32, 64, 128, and 256 entries obtains the increasing coverage of predictable operations. Using the value predictor with 256 entries collects almost all dynamic predictable operations in a program. Moreover, the recorded operations by hardware-based value profiling have very high value prediction accuracies of 94% on an average. This shows that hardware-based value profiling is accurate to identify highly predictable operations. The VSS optimization is also experimented based on the feedbacks from hardware-based value profiling, and yields the speedups of up to 19% and averaging 3.5%.

Chapter 7

Conclusions and Future work

This thesis has proposed compiler-driven value speculation scheduling to exploit the predictability for the values generated by register-writing operations to improve the performance of microprocessors. The value speculation scheduling (VSS) technique leverages advantages of both hardware schemes for value prediction and compiler schemes for exposing ILP. Two new predicting and updating operations, LDPRED and UDPRED, are designed to be the interface between value prediction hardware and program code. The VSS algorithm utilizes LDPRED and UDPRED operations to break critical paths in a program to shorten execution time. Future work will include the investigation of new applications that utilize LDPRED and UDPRED operations to improve the performance. The correlation between different operations can improve the prediction accuracies of operations. The LDPRED and UDPRED operations can be re-designed by using register values as the indices to access the value predictor. Thus, control correlation of program flow and value correlation of linked data structures can be captured to enhance the predictability of operations.

To improve the techniques for value speculation, the value speculation model has been proposed as solving an optimal edge selection problem in a data dependence graph. An efficient algorithm has been designed based on three properties observed from the optimal edge selection problem. The selected dependences are then exposed to the hardware or the compiler to obtain maximal benefits from value speculation. In future work, the value speculation model can be broadened to target not only register flow dependences but also memory dependences between load and store operations. The integrated model can serve as a compilation phase of benefit analysis to select flow dependences via value speculation [22] and memory dependences via data speculation [1], [41].

Without any modification to the hardware, software-only value speculation scheduling (SVSS) has been proposed to improve the performance of existing microprocessors. Significant speedups have been shown for using the software static stride value predictor to optimize the SPECint95 programs. Future work will include the design of new software value predictors to predict operations with different patterns, so that the SVSS scheme can optimize more predictable operations.

Hardware-based value profiling has been investigated to accurately collect highly predictable operations at run-time for reducing the overhead of program profiling and eliminating the need of profile training inputs. The VSS optimization has been experimented based on the feedbacks from hardware-based value profiling. In future work, the results of hardware-based value profiling can assist dynamic optimization [38].

The utilization of hardware-based value profiling and the invocation of performing the VSS optimization can be integrated together to improve the performance dynamically.

In this thesis, the VSS and SVSS schemes have been experimented on VLIW architectures and have showed encouraging speedups in the SPECint95 benchmarks. Besides statically-scheduled machines, the VSS and SVSS optimizations can be applied to dynamically-scheduled machines as well. Future work will experiment VSS and SVSS on different architectures to investigate the effectiveness of the techniques. Overall, this thesis provides a promising way of applying value prediction and value speculation to future microprocessors.

References

[1] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhall, W. W. Hwu, "Dynamic Memory Disambiguation Using the Memory Conflict Buffer," *Proceedings of the 6th International Conference on Architecture Support for Programming Languages and Operating Systems,* October 1994.

[2] M. H. Lipasti, C. B. Wilkerson, J. P. Shen, "Value Locality and Load Value Prediction," *Proceedings of the 7th International Conference on Architecture Support for Programming Languages and Operating Systems*, October 1996.

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

[4] F. Gabbay and A. Mendelson, "Can Program Profiling Support Value Prediction?" *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997.

[5] F. Gabbay and A. Mendelson, "The Effect of Instruction Fetch Bandwidth on Value Prediction," *Proceedings of the 25th International Symposium on Computer Architecture,* June 1998.

[6] F. Gabbay, "Speculative Execution based on Value Prediction," EE Department TR #1080, Technion, November 1996.

[7] B. Calder, P. Feller, and A. Eustace, "Value Profiling," *Proceedings of the 30th International Symposium on Microarchitecture,* December 1997.

[8] B. Calder, G. Reinman, and D. Tullsen, "Selective Value Prediction," *Proceedings of the 26th International Symposium on Computer Architecture*, May 1999.

[9] E. Tune, D. Liang, D. Tullsen, and B. Calder, "Dynamic Prediction of Critical Path Instructions," *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, January 2001.

[10] Y. Sazeides and J. E. Smith, "The Predictability of Data Values," *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997.

[11] Y. Sazeides and J. E. Smith, "Implementation of Context Based Value Predictors," Technical Report ECE-97-8, University of Wisconsin-Madison, December 1997.

[12] Y. Sazeides and J. Smith, "Modeling Program Predictability", *Proceedings of the 25th International Symposium on Computer Architecture*, June 1998.

[13] K. Wang and M. Franklin, "Highly Accurate Data Value Prediction using Hybrid Predictors," *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997.

[14] R. Sathe and M. Franklin, "Available Parallelism with Data Value Prediction", *Proceedings of the 5th International Conference on High Performance Computing*, December 1998.

[15] M. Burstscher and B. G. Zorn, "Exploring Last n Value Prediction," *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, October 1999.

[16] T. Nakra, R. Gupta, and M. L. Soffa, "Global Context-Based Value Prediction," *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, January 1999.

[17] W. A. Havanki, S. Banerjia, and T. M. Conte, "Treegion Scheduling for Wide-Issue Processors," *Proceedings of the 4th International Symposium on High Performance Computer Architecture*, February 1998.

[18] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The Superblock: An Effective Technique for VLIW and Superscalar Compilation," *The Journal of Supercomputing*, January 1993.

[19] S. Aditya, V. Kathail, and B. R. Rau, "Elcor's Machine Description System: Version 3.0," Hewlett-Packard Laboratories Technical Report HPL-98-128, October 1998.

[20] V. Kathail, M. Schlansker, and B. R. Rau, "HPL-PD Architecture Specification: Version 1.1," Hewlett-Packard Laboratories Technical Report HPL-93-80 (R.1), February 2000.

[21] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, “Introduction to Algorithms,” MIT Press, 1990.

[22] C. Fu, M. D. Jennings, S. Y. Larin, and T. M. Conte, “Value Speculation Scheduling for High Performance Processors,” *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems,* October, 1998.

[23] C. Fu, M. D. Jennings, S. Y. Larin, and T. M. Conte, “Software-Only Value Speculation Scheduling,” Technical Report, Department of Electrical and Computer Engineering, North Carolina State University, June 1998.

[24] C. Fu and T. M. Conte, “Value Speculation Mechanisms for EPIC Architectures,” Technical Report, Department of Electrical and Computer Engineering, North Carolina State University, October 1998.

[25] T. Nakra, R. Gupta, and M. L. Soffa, “Value Prediction in VLIW Machine,” *Proceedings of the 26th International Symposium on Computer Architecture*, May 1999.

[26] E. Larson and T. Austin, “Compiler Controlled Value Prediction Using Branch Predictor Based Confidence,” *Proceedings of the 33rd International Symposium on Microarchitecture*, December 2000.

[27] L. Gwennap, “Intel, HP Make EPIC Disclosure,” *Microprocessor Report*, 11(14): 1-9, October 1997.

[28] C. Dulong, “The IA-64 Architecture at Work,” *Computer,* vol. 31 no.7, July 1998.

[29] L. Gwennap, “MIPS R10000 uses decoupled architecture,” Microprocessor Report, 8(14):18-22, October 1994.

[30] L. Gwennap, “Digital 21264 Sets New Standard,” Microprocessor Report, 10(14):11-16, October 1996.

[31] Intel Corporation, “Itanium Software Conventions and Runtime Architecture Guide,” (Available from http://developer.intel.com/design/ia64/downloads/245358.htm), September 2000.

[32] D. W. Wall, “Global Register Allocation at Link Time,” Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction, June 1986.

[33] T. M. Conte, S. Banerjia, S. Y. Larin, K. N. Menezes, and S. W. Sathaye, “Instruction fetch mechanisms for VLIW architectures with compressed encodings,” *Proceedings of the 29th International Symposium on Microarchitecture*, December 1996.

[34] K. N. Menezes, S. W. Sathaye, and T. M. Conte, “Path prediction for high issue-rate processors,” *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques*, November 1997.

[35] J. Hoogerbrugge, “Dynamic Branch Prediction for a VLIW Processor,” *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, October 2000.

[36] T. Ball and J. R. Larus, “Branch Prediction for Free,” *Proceedings of the ACM SIGPLAN ‘93 Conference on Programming Language Design and Implementation*, June 1993.

[37] K. N. P. Menezes, “Hardware-based Profiling for Program Optimization,” Ph.D. Thesis, Department of Electrical and Computer Engineering, North Carolina State University, 1997.

[38] S. W. Sathaye, “Evolutionary Compilation for Code Compatibility and Performance,” Ph.D. Thesis, Department of Electrical and Computer Engineering, North Carolina State University, 1998.

[39] P. P. Chang, S. A. Mahlke, and W. W. Hwu, “Using Profile Information to Assist Classic Code Optimizations,” *Software-Practice and Experience*, vol. 21, December 1991.

[40] T. M. Conte, B. A. Patel, and J. S. Cox, “Using Branch Handling Hardware to Support Profile-Driven Optimization,” *Proceedings of the 27th International Symposium on Microarchitecture*, November 1994.

[41] Intel Corporation, “IA-64 Application Developer’s Architecture Guide,” (Available from http://developer.intel.com/design/ia64/downloads/adag.htm), May 1999.