ABSTRACT

ZHOU, HUIYANG

Using Performance Bounds to Guide Code Compilation and Processor Design

(Under the direction of Professor Thomas M. Conte)

Performance bounds represent the best achievable performance that can be delivered by target microarchitectures on specified workloads. Accurate performance bounds establish an efficient way to evaluate the performance potential of either code optimizations or architectural innovations.

We advocate using performance bounds to guide code compilation. In this dissertation, we introduce a novel bound-guided approach to systematically regulate code-size related instruction level parallelism (ILP) optimizations, including tail duplication, loop unrolling, and if-conversion. Our approach is based on the notion of code size efficiency, which is defined as the ratio of ILP improvement over static code size increase. With such a notion, we (1) develop a general approach to selectively perform optimizations to maximize the ILP improvement while minimizing the cost in code size, (2) define the optimal tradeoff between ILP improvement and code size overhead, and (3) develop a heuristic to achieve this optimal tradeoff.

We extend our performance bounds as well as code size efficiency to perform code-size-aware compilation for real-time applications. The profile independent performance bounds are proposed to reveal the criticality for each path in a task. Code optimizations can then focus on the critical paths (even at the cost of non-critical ones) to

reduce the worst-case execution time, thereby improving the overall schedulability of the real-time system.

For memory intensive applications featuring heavy pointer chasing, we develop an analytical model based on performance bounds to evaluate memory latency hiding techniques. We model the performance potential of these techniques and use the analytical results to motivate an architectural innovation, called recovery-free value prediction, to enhance memory level parallelism (MLP). The experimental results show that our proposed technique improves MLP significantly and achieves impressive speedups.

USING PERFORMANCE BOUNDS TO GUIDE CODE COMPILATION AND PROCESSOR DESIGN

by

HUIYANG ZHOU

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

2003

APPROVED BY:

Prof. Thomas M. Conte Chair of Advisory Committee	Prof. Gregory T. Byrd
Prof. S. Purushothaman Iyer	Prof. Eric Rotenberg

BIOGRAPHY

Huiyang Zhou was born in Xi'an, P.R. China. He received his Bachelor of Engineering degree and Master of Engineering degree from Xian Jiaotong University, P. R. China in 1992 and 1995 respectively. He went on to join National University of Singapore, Singapore in 1996. After finishing his second Master's degree, he joined North Carolina State University, USA in 1998 as a Ph.D. student in the Computer Engineering program. In May 2000, he became a member of TINKER research group under the supervision of Dr. Thomas M. Conte. He completed his Ph.D. degree in the summer of 2003 and is now an assistant professor in the Computer Science department at the University of Central Florida, USA.

ACKNOWLEDGEMENTS

First of all, I would like to express my deep appreciation to my Ph.D. advisor, Professor Thomas M. Conte. Tom has been a great advisor and his vision has led me through my graduate research career. Without his guidance and encouragement, this dissertation would not even be possible. Also, I would like to thank Prof. Thomas M. Conte, Prof. Eric Rotenberg, Prof. Gregory T. Byrd, and Prof. S. Purushothaman Iyer for serving on my dissertation committee. Special thanks to Prof. Eric Rotenberg for many inspiring discussions and his insightful advice.

I would like to thank all the previous and current members of the TINKER research group, Chao-ying Fu, Emre Ozer, Sergei Larin, Matt Jennings, Mark Toburen, Kim Hazelwood, Vikram Rao, Tripura Ramesh, Jill Bodine, Fei Gao, Ugur Gunal, and Saurabh Sharma for their contributions to the LEGO compiler and the simulation framework that I have used in my dissertation research. I have enjoyed the collaboration in our group very much.

I would like to thank my wife, Rong Wang, my mother, Ruiying Zhang, my father, Jianqun Zhou, and my older brother, Chunyang Zhou, for their love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1 Introduction

1.1 Introduction

Performance bounds represent the best achievable performance that can be delivered by target architectures on specified workloads. Previous works [8],[52],[53] proposed the use of performance bounds to evaluate different architectures by measuring how closely the achieved performance compares to the performance bounds.

In this dissertation, we advocate using performance bounds to guide code optimizations and processor design. The insight is that since performance bounds reflect the best achievable performance, the difference between two sets of performance bounds, one for the original and one for the optimized workload or architecture, simply reveals the performance potential of such optimizations.

In code compilation, instruction optimizations can lead to different outcomes in performance and compiler efficiency depending on where these optimizations are applied. In order to use code optimizations efficiently and judiciously, two major issues need to be addressed. First, an effective cost-benefit model is needed so that the performance gains can be analyzed before actually performing time-consuming

instruction scheduling or the optimization itself. Secondly, a systematic method is needed to selectively apply various types of optimizations based on the cost model. Performance bounds serve this purpose appropriately, as they enable efficient measurement of the performance limit of an optimization and also help us to understand the bottlenecks when the performance potential is not fully achieved.

Two sets of tight performance bounds are proposed in this dissertation for different applications. *Profile-guided performance bounds* are based on edge profile information, and *profile-independent performance bounds* reveal the criticality of different control paths in terms of worst-case execution time (WCET) in real-time applications. The proposed performance bounds are used to guide code compilation, *code-size-aware compilation* in particular.

Current microprocessors exploit instruction level parallelism (ILP) aggressively to achieve high performance. Therefore, ILP optimizations such as tail duplication, loop unrolling, and if-conversion, are commonly used in code compilation to boost the ILP of the program. However, these optimizations usually involve significant static code size increases, thus raising concerns about the effects on instruction cache (I-cache) and instruction translation lookaside buffer (I-TLB) performance. For embedded systems, the cost of memory for storing the static code is also an important factor. Another issue with oversized programs is the compilation time, since compilation complexity is usually $O(N^2)$, where N is the number of instructions in the function/program. In order to achieve a good trade-off between performance improvement and code size increase, we introduce a systematic approach to regulate code size related ILP optimizations so that the performance gains are maximized at a very small cost in static code size increase. Our

approach is based on the notion of code size efficiency, defined as the ratio of ILP improvement over the static code size increase. Based on such a notion, we (1) develop a general approach to selectively perform optimizations to maximize the ILP improvement at a minor cost in code size, (2) define the optimal tradeoff between the ILP improvement and the code size overhead, and (3) develop a heuristic to achieve this optimal tradeoff. Since profile-guided performance bounds are used to evaluate the ILP improvement, our algorithms have the advantage of low computational complexity, which is important to the compile time of the program. Experiments using the SPEC CINT 2000 benchmarks [30] show that performance improves significantly with very little code size increase using our systematic method for regulating code transformations. The results also show that our simple heuristic is both effective and robust in achieving the optimal tradeoff.

In real-time applications, the major concern is to finish tasks within specified deadlines. We advocate using code optimizations as well as instruction scheduling to reduce the worst-case execution time (WCET) of each task, thereby increasing the overall system-level schedulability. With such an objective, the measure of code size efficiency is extended with profile-independent performance bounds so that it reflects how much the WCET is potentially reduced when additional instructions are introduced from various code optimizations. Then, a similar approach to regulate ILP optimizations is developed to selectively perform these optimizations so that the WCET is significantly reduced with small static code size increases.

With great effort from both the compiler and hardware, current microprocessors have a tremendous capability to exploit ILP aggressively to achieve high performance computation. However, due to the comparably slower speed of the memory, if the

computation involves a slow memory operation (e.g., a cache miss), the execution pipeline of a microprocessor usually has to be stalled in order to wait for the required data to be fetched from memory. For memory intensive workloads, the slow memory accesses form the critical path of the program and dominate the overall execution time. For such workloads, especially irregular programs with heavy pointer chasing, reducing or hiding the memory access latencies is essential to achieve high performance and has been an active research topic.

In this dissertation, we propose an analytical model to bound the performance potential of two different, yet related memory access latency hiding techniques, namely address prediction based memory prefetching [14],[33] and value prediction [43],[44],[21]. Interesting insights are revealed from our analytical model for either technique and the code characteristics are identified for which one technique outperforms the other. It is found that value prediction is a very powerful technique to improve memory-level-parallelism (MLP) for future high performance microprocessors. One key reason is that while prefetching only brings the data close to the microprocessor, value prediction takes one step further by using the fetched data to drive the dependent missing loads to be executed. If the prediction is correct in the first place, such speculative execution propagates the predictability even though the dependent loads could be unpredictable. Such observations also motivate an innovation, called *recovery-free value prediction*, to improve MLP more cost-effectively.

In recovery-free value prediction, value speculation is used only for prefetching so that the complex prediction validation and misprediction recovery mechanisms used in traditional value prediction schemes are avoided, as well as the associated recovery

penalties from value misprediction. Only minor changes in the microarchitecture are needed to implement recovery-free value prediction, and the same hardware modifications also enable speculative memory disambiguation for prefetching. Another advantage is that recovery-free value prediction uses the actual execution results rather than execution results based on previous predictions to update value predictors, thereby achieving better prediction results. The experiments show that our proposed technique enhances MLP effectively and achieves significant speedups even with a simple stride value predictor.

1.2 Contributions of the Dissertation

This dissertation addresses several important issues in high performance computer architecture. First, tight profile-guided performance bounds are proposed and a quantitative measure of code size efficiency is proposed using such performance bounds. Based on this measure, algorithms with low computational complexity are designed to selectively perform different ILP code optimizations.

Secondly, we extend performance bounds to guide code compilation to reduce the WCET for timing critical tasks in real-time applications. The revised code size efficiency reflects the WCET reduction at the cost of code size increase resulting from a code optimization. A similar algorithm to the one proposed for profile-guided compilation is developed to selectively perform code optimization to reduce the WCET aggressively at a minor cost in static code size increase.

Thirdly, an analytical model is proposed to evaluate the memory latency hiding techniques including address prediction based prefetching and value prediction. Key insights are revealed from the model to guide both the compiler and processor design.

Fourthly, we propose a novel approach, called recovery-free value prediction to enhance MLP. Our approach has low hardware complexity and achieves significant speedups for a range of memory intensive benchmarks with a simple value predictor.

1.3 Outline of the Dissertation

The dissertation is organized as follows. Chapter 2 presents the proposed performance bounds, including both profile-guided and profile-independent bounds. The code compilation for code size efficiency using profile-guided performance bounds is in Chapter 3. Chapter 4 discusses code-size-aware compilation for real-time applications using profile-independent performance bounds. The performance modeling of memory hiding techniques is contained in Chapter 5. Chapter 6 presents our proposed recovery-free value prediction mechanism to enhance memory level parallelism. Chapter 7 concludes the dissertation and addresses further research directions.

Chapter 2 Performance Bounds

In this chapter, we first discuss the previous work on performance bounds in Section 2.1. Section 2.2 contains a brief background description of treegion-based instruction scheduling, which is the instruction-scheduling framework used in this work. Then, we introduce our proposed profile-guided performance bounds in Section 2.3 and profile-independent performance bounds in Section 2.4.

2.1 Previous Work

As discussed in Chapter 1, performance bounds were previously proposed to evaluate different computer architectures. In [53], a set of performance bounds are presented based on scientific workloads to evaluate a range of computer architectures. For vector processors including Cray-1, Cray-XMP, and Cray-2, Tang and Davison [67] used vectorizable Livermore Fortran Kernels (LFK) as the target workloads and developed a simple bound model assuming that the function unit bandwidth is the bottleneck. Then, an accurate timing model was developed to determine the achieved performance. When achieved performance lagged the performance bound, an ad-hoc approach was used to identify the performance inhibitors. Mangione-Smith, Abraham and

Davison [52],[54] extended the workload to include Livermore Fortran kernels that are not vectorizable and generalized their processor models to study processors such as the Astronautics ZS-1, MIPS R3000 and IBM RS/6000. The performance bound was modified to be the maximum of the resource bound and the dependence bound. The resource bound is basically the bandwidth requirement of the workload. The dependence bound accounts for the loop-carried dependence for non-vectorizable loops.

Boyd and Davison [8],[9] further extended the above simple bound models to a hierarchical performance model (MACS) in order to study the performance bottlenecks in a more formal way. Such a hierarchical model captures the performance impact of following factors: machine architecture, application workload, high-level compiler optimization, instruction selection, and instruction scheduling. The M (machine) bound is the peak performance that the processor can provide. The MA (machine-application) bound considers the workload requirement. The MAC (machine-application-compiler) bound improves the MA bound by counting the actual operations in the compiled workload. The MACS (machine-application-compiler-scheduler) further refines the performance bound by including the instruction scheduler impact. By measuring the performance gap between different bounds, this hierarchical bound model is shown to be very helpful in identifying the performance losses in the spectrum from compilation to the target architecture.

Performance bounds are also proposed to guide instruction scheduling and hardware synthesis. Tight lower bounds of basic block (BB) scheduling proposed in [37],[60] were used to prune the design space of hardware synthesis. Based upon the tight bounds of BB, Eichenberger and Meleis [19] computed a tight lower bound of a

superblock using pairwise bounds to account for resource conflicts among branches. Such bounds are then used as a heuristic to schedule operations at each cycle, and these bounds are also updated to reflect the schedule decisions during the scheduling process. Although our performance bound computation is based on a different type of scheduling region, the treegion, the bound calculation is similar to these previously introduced bounds since all these bounds are trying to capture the data dependence and resource constraint impact.

Compared to these previous works on performance bounds, our use of performance bounds is different in that we propose to use the change/reduction in performance bounds as a fast and accurate way to capture the performance potential of either a code optimization or a hardware innovation on specified workloads based on target microarchitectures. We describe our proposed performance bounds in Section 2.3 and Section 2.4. Before that, a brief overview of our compiler framework, the treegion-based global acyclic instruction scheduler in particular, is presented in the next section.

2.2 Treegions and Treegion-based Global Instruction Scheduling

Treegion-based global scheduling [28],[78] is used as the acyclic scheduling framework in this dissertation. However, it needs to be pointed out that although the experimental results are obtained using treegion scheduling, the same methodology of the performance bound as well as the code size efficiency study in next two chapters is applicable to other global scheduling approaches, such as superblock scheduling [31] and hyperblock scheduling [51].

Treegion-based global scheduling aims for high performance for wide issue VLIW / EPIC processors although it can be applied to superscalar processors as well. It has two steps: treegion formation and tree traversal scheduling (TTS). A treegion is a single-entry / multiple-exit nonlinear code region that consists of basic blocks (BBs) with the control-flow forming a tree, as illustrated in Figure 2.1.

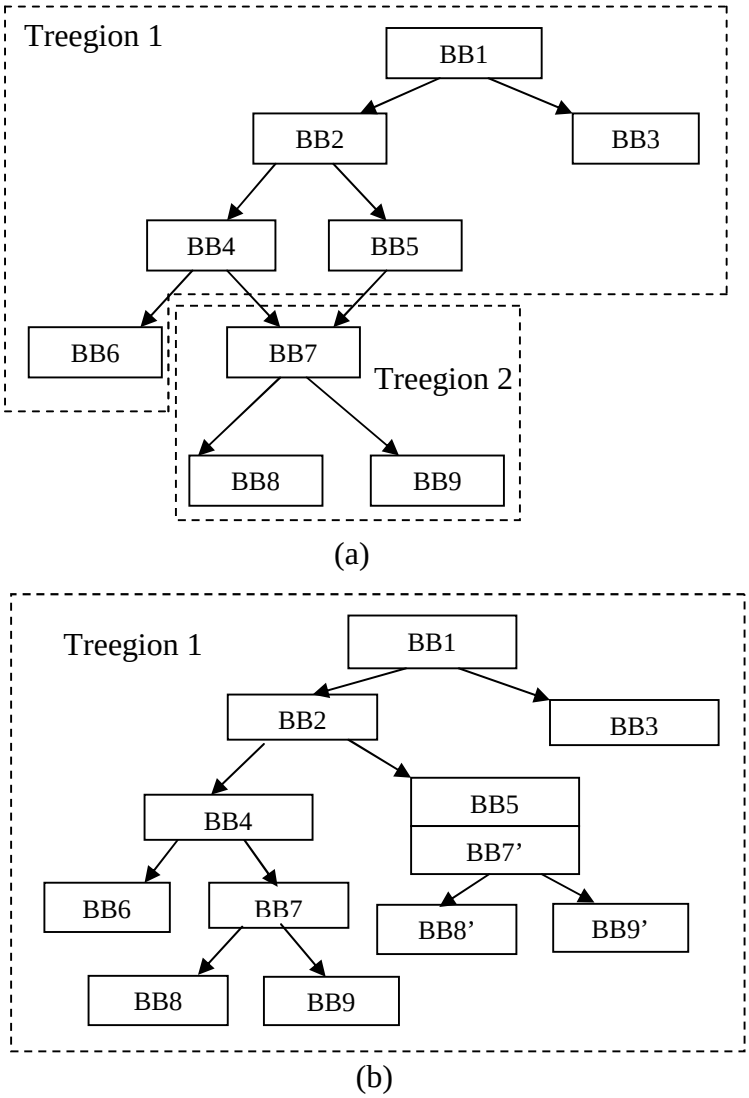


Figure 2.1 (a) The CFG and the natural treegion construction; (b) The treegion constructed after tail duplication.

Based on the control flow graph (CFG) in Figure 2.1a, two treegions are formed. The treegions that are formed without any tail duplication are referred to as *natural treegions*. When tail duplication is applied, a larger treegion can be formed. For the example CFG in Figure 2.1a, after BB7, BB8, and BB9 are duplicated and the corresponding unconditional branches are removed, one treegion is formed containing all the BBs in the CFG, as shown in Figure 2.1b. Such duplication enables speculation from BB7, BB8, BB9 and their duplicates, thereby increasing ILP. The trade-off for exposing ILP through treegion formation is the code-expansion that results from duplicates of BB7, BB8 and BB9. Note that in this dissertation, tail duplication is performed on the unit of the natural treegion (i.e., merge points), e.g., in the example of Figure 2.1, the entire treegion 2 is duplicated instead of BB7. In the previous treegion scheduling works, tail duplication is performed based on a heuristic discussed in [28], which we refer to as Havanki's heuristic and briefly describe as follows. Havanki's tail duplication heuristic is based on several factors: code expansion limit, path count (the number of paths in a treegion) and the number of incoming edges to a merge point. The code expansion limit is a global control parameter, while the other two are based on the topology of the CFG. When any of these limits is reached, tail duplication will stop and a new treegion will be formed. The advantage of this heuristic is that it solely depends on the topology of the CFG and it is not susceptible to profiling errors. But it does not take the performance impact of such duplication into account. As will be seen in Chapter 3, we develop an integrated approach to perform selective tail duplication, loop unrolling and if-conversion, and achieve much higher ILP improvements [75].

During tree traversal scheduling (TTS), the BBs in a treegion are scheduled in a predetermined traversal order based on treegion topology and profile information. When a BB is currently being scheduled, those instructions that are dominated by the BB will be considered as scheduling candidates until the block-ending branch is scheduled. In this way, speculation is enabled from all the paths starting from the BB. Those candidate operations are scheduled based on an order determined by a heuristic that includes their execution frequency, exit count, and data dependence height. The details of tree traversal scheduling can be found in [78].

2.3 Profile-Guided Performance Bounds

Due to complexity, many compiler frameworks partition a function body into many multi-path regions and each region is used as a scheduling unit. We establish a lower bound of execution time for such a single-entry multiple-exit region since instructions are rarely moved across the scheduling region boundary. Performing bound computation at the granularity of the scheduling region is important as it captures global instruction scheduling impacts accurately. With region-level performance bounds, we can derive easily the bounds at the procedure/function level and the program level.

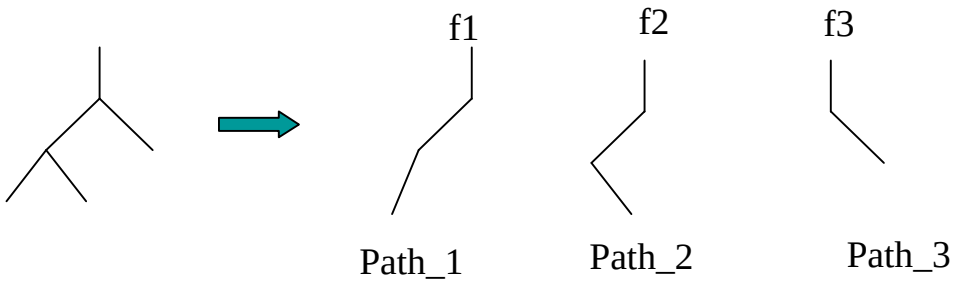


Figure 2.2. A CFG example containing three control paths.

For a single-entry multiple-exit region, if execution frequency for each control path is determined from profile information, we can compute the lower bound of execution time (LBET) as a weighted sum of LBET of each path. For the example control flow graph (CFG) shown in Figure 2.2, there are three control paths in the region and the execution frequencies f1, f2, and f3 are associated with each path respectively. So, the lower bound execution time of this region can be computed as the sum of the LBET of each path weighted by its execution frequency. We can write this weighted sum as the following equation.

$$LBET = \sum_{path\_i} LBET_{path\_i} * freq_{path\_i}$$
$$= \sum_{path\_i} Max\left(data\_dependence\_bound_{path\_i},\ resource\_bound_{path\_i}\right) * freq_{path\_i}$$

Equation 2-1

In Equation 2-1, the *lower bound of execution time* (LBET) of a region is a weighted sum of the LBET of each path, which is in turn computed as the maximum of the *data dependence bound* and the *resource bound* of the path. True data dependence height of Data Dependence Graph (DDG) is used as the data dependence bound assuming software renaming is available at schedule time to remove false register dependencies. By calculating the data dependence bound along each path, the potential control speculation effect is considered implicitly as the control dependence is not enforced. Also, using only the true data dependence simplifies the bound computation since it is an $O(N)$ computation.

Resource bound in Equation 2-1 is calculated similar to the ResMII (resource-constrained minimum-initiation-interval) calculation in iterative modulo scheduling [59], as follows.

$$resource\_bound_{path\_i} = \underset{k}{Max}\lceil (Num\_Insn_k / Num\_FU_k) \rceil \qquad \textbf{Equation 2-2}$$

In Equation 2-2, Num_Insn_k represents the number of operations that use the function unit type *k*. Num_FU_k represents the number of function units of type *k* available in the processor. The ratio (ceil) of these two numbers shows the resource constraints of function units of type *k*. Then, resource bound is calculated as the maximum constraint of all types of function units. From our experience, load/store units and branch units are usually critical resources for most integer benchmarks in the SPEC 2000 Integer benchmark suite.

Again, the execution frequency for each path, $Freq_{path\_i}$, used as the weight of the corresponding path in Equation 2-1, is obtained from edge profiling.

Since the LBET of each scheduling region describes its execution time, the LBET of the whole program is simply the sum of the LBETs of all the scheduling regions. So, we can compute the program-level performance bound with the following steps:

1. Forming the scheduling regions, such as treegions or superblocks, based on the control flow graph.
2. Compute LBET of each region and take the summation as the LBET of the program.

Next, we use a simple code example to show the profile-guided bound calculation and the relationship between the bound tightness and the region type. As each different region type defines a different scheduling scope, the calculated bounds can be different

for the same code segment. A code example in IA-64 [32] style assembly is given in Figure 2.3, and the corresponding C code is shown in Figure 2.4.

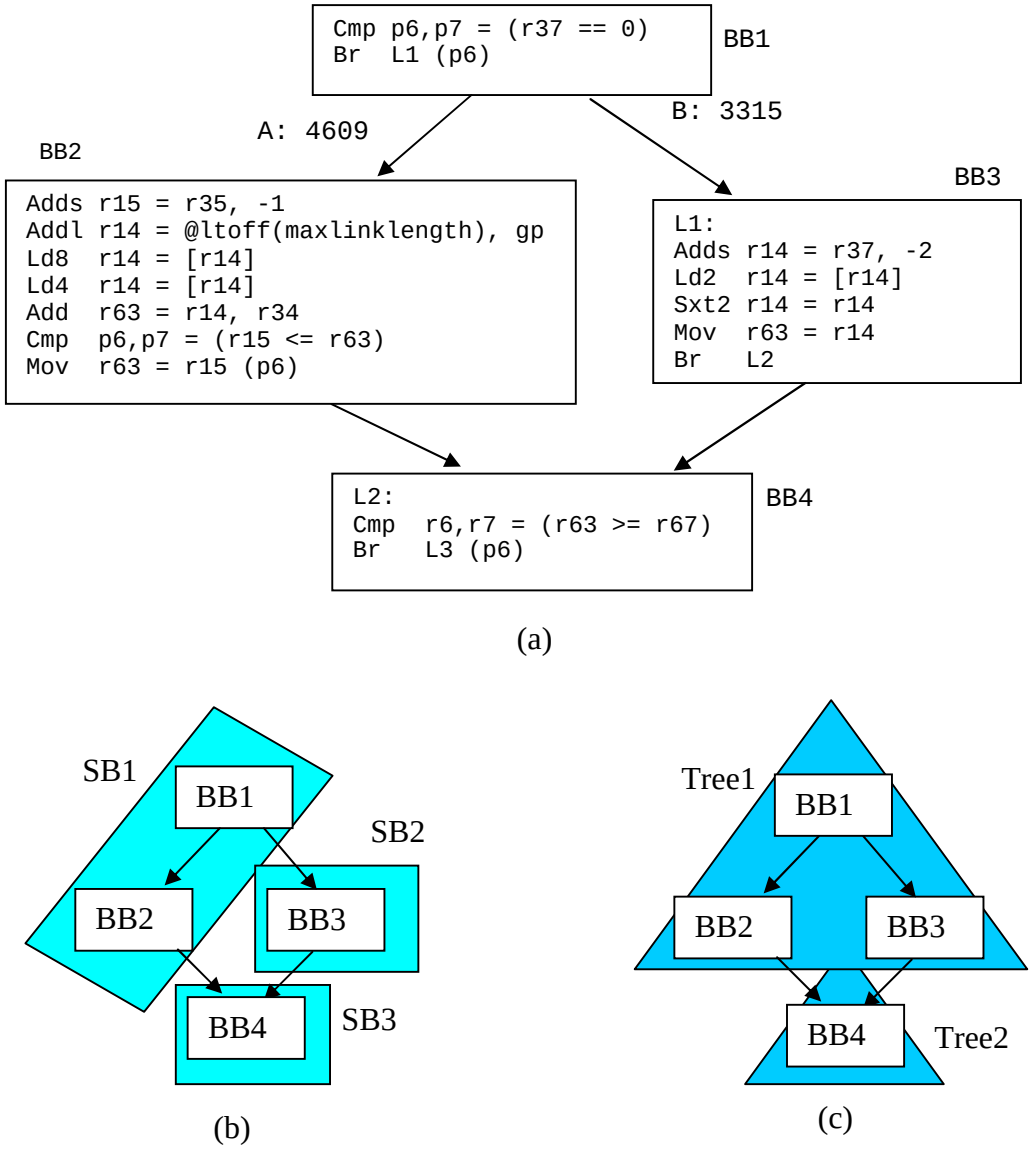


Figure 2.3. (a) Code segment from the benchmark *parser* (function list_link). Numbers along the edge labels are edge profiles; (b) The superblock formed without tail duplication; (c) The natural treegion formed.

As shown in Figure 2.3a, the code segment is a simple diamond structure extracted from the benchmark *parser*. First, we show the bound computation for superblock scheduling. If no code expansion optimization is performed, three superblocks

(SB) are formed for the code segment: SB1 contains BB1 and BB2, SB2 contains BB3, and SB3 contains BB4, as shown in Figure 2.3b. Assuming our machine model has the following configuration: 6-wide issue (2 ALU, 2 ALU/LD/ST, 2 ALU/BR, e.g., Itanium-I and II); load operations have a 2-cycle latency and all other integer operations have a 1-cycle latency (except CMP instructions which can be issued at the same cycle as the consuming branch). We can compute the lower bound of execution time (LBET) of SB1 using Equation 2-1 as: 1*3315 + 8*4609 = 40,187 cycles; LBET of SB2 as 5 * 3315 = 16,575 cycles; and LBET of SB3 as 1 * (3315 + 4609) = 7,924 cycles. The performance bound of the hammock is the sum of the bounds of these superblocks (64,686 cycles).

```
…
if(re == NULL) {
    end_word = MIN(rw-1, lw+maxlinklength);
} else {
    end_word = re->word;
}
…
```

Figure 2.4. The corresponding *C* code of the assembly code segment in Figure 2.3. The global variable *maxlinklength* is accessed through a linkage table.

Next, if we use treegions as basic scheduling regions, two natural treegions can be formed for this code example without any code replication: Tree1 contains BB1, BB2, and BB3; and Tree2 contains BB4, as shown in Figure 2.3c. For the same machine model, the LBET of Tree1 is computed as: 4609*8 + 3315*5 = 53,447 cycles; the LBET of Tree2 is 1 * (3315 + 4609) = 7,924 cycles. The LBET of the hammock is the sum of the LBETs of Tree1 and Tree2 (61,371 cycles). Compared to the LBET computed using superblocks, the treegion-based LBET is smaller as it considers the possibility of control speculation not only from BB2 to BB1 but also from BB3 to BB1. The superblock-based

approach, however, considers speculation only from BB2 to BB1. From this example, it can be seen that the performance bounds also reveal the potential of a particular instruction-scheduling algorithm and it illustrates that treegion scheduling provides better scheduling capabilities by enabling speculation from multiple execution paths.

The region expansion optimization, duplication of BB4 in this example, could potentially reduce the LBET of both superblock-based LBET and treegion-based LBET and this will be discussed in detail in Chapter 3 as we evaluate the performance impact of code size related optimizations.

2.4 Profile Independent Performance Bounds

For real-time applications, the most important objective is to guarantee that a task finishes by a specified deadline instead of reducing the average execution time. As a result, the worst-case execution time (WCET) is commonly used assuming a program will experience its longest control flow path. As our objective is to evaluate WCET reduction of ILP optimizations for real-time applications, we propose a profile-independent bound for a single-entry multiple-exit region as follows:

$$LBWT = \underset{path\_i}{Max} LBET_{path\_i}$$
$$= \underset{path\_i}{Max}\left(Max(data\_dependence\_bound_{path\_i},\quad resource\_bound_{path\_i})\right)$$

Equation 2-3

As shown in Equation 2-3, the lower bound of WCET (LBWT) for a multi-path region is the maximum of the lower bound execution time (LBET) of each path, which is computed as the maximum of the *data dependence bound* and the *resource bound* of the path, as described before in Section 2.3. In other words, the worst case control flow (i.e.,

the path with longest execution time) is assumed while the lower bound of execution time is used for each path. Since such a lower bound is used, the actual execution time along the path could potentially exceed this lower bound. So, it apparently conflicts the purpose of worst-case execution time. However, remember that we use LBWT to measure the impact of WCET reduction due to code optimizations instead of using LBWT directly as the final WCET measure. Measuring the actual execution time along each path requires the scheduled code. It is unacceptable in practice since time-consuming instruction scheduling needs to be performed in order to measure the impact for every single code optimization instance. Using LBET for each path, on the other hand, provides an accurate estimate of the actual execution time and associates low computational complexity. Moreover, this LBWT can be used to check the soundness of the deadline setting: if the predetermined deadline exceeds the LBWT, it is impossible that the task can be finished in time when the longest control path is taken. In such a case, the system has to reassign the deadlines, adopt a more powerful processor, or optimize the code more aggressively.

Computing *LBWT at the function level* is complicated due to complex CFGs and multiple regions in a function body. Although we can use a simple approach, such as taking the summation of the LBWTs of each region as the LBWT for the function, the computed bounds are overly pessimistic as many impossible control paths are assumed.

Here, we use a similar approach to the static WCET analysis for scheduled code, in which the analyzer derives WCET for each path, then for loop bodies, and finally for functions in the program. The WCET of the *main* function is simply the WCET for the entire program. Compared to this path-based static analysis approach for scheduled code, treegion-based LBWT provides an efficient way to incorporate the instruction scheduling

effect accurately, especially the control speculation effect, and limits the enumeration of the possible control paths. Next, we use an example to derive this treegion-based LBWT analysis. We start with an innermost loop body, which may contain more than one treegion. One such example is shown in Figure 2.5.

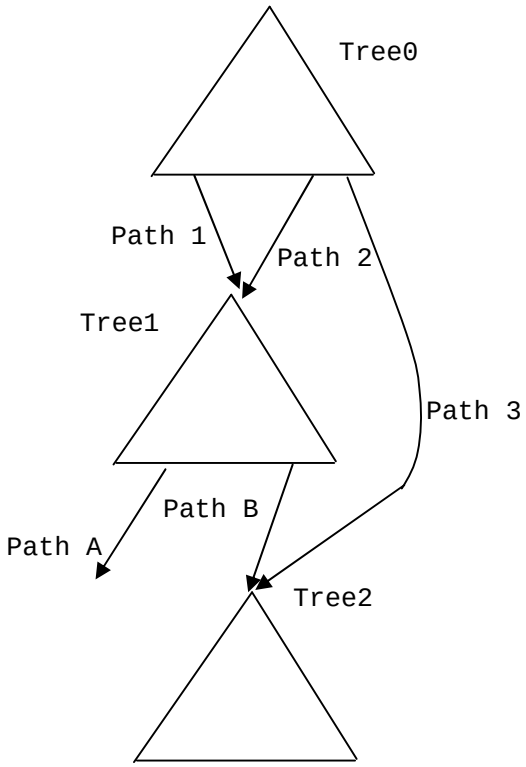


Figure 2.5. Deriving LBWT in a complex CFG without loops.

The CFG in Figure 2.5 contains three treegions. In order to compute the LBWT of such a code segment, we extend Equation 2-3 to Equation 2-4 to compute the LBWT for each treegion. The LBWT for treegion 0 in this example is the LBWT for the overall code segment.

$$LBWT = Max(LBET_{path\_1}+LBWT_{base\_path\_1}, \ldots, LBET_{path\_k}+LBWT_{base\_path\_k})$$

Equation 2-4

In Equation 2-4, LBWT of a treegion is computed as the maximum LBWT of every path in the treegion, which is in turn defined as the sum of the LBET of the path ($LBET_{path\_i}$) and the LBWT of the treegion that the path leads to ($LBWT_{base\_path\_i}$). The term $LBET_{path\_i}$ is defined as before, i.e., the maximum of the data dependence bound and the resource bound. The term $LBWT_{base\_path\_i}$ is computed recursively using Equation 2-4 based on the control dependence relationship among treegions. For exit paths or return paths, $LBWT_{base}$ is zero. For the code example in Figure 2.5, the overall LBWT (i.e., LBWT of treegion 0) is computed as follows:

$$LBWT_{treegion0} = Max(LBET_{path\_1}+LBWT_{base\_path\_1}, \ldots, LBET_{path\_k}+LBWT_{base\_path\_k})$$

$$= Max(LBET_{path\_1}+LBWT_{treegion1}, LBET_{path\_2}+LBWT_{treegion1}, LBET_{path\_3}+LBWT_{treegion2}).$$

LBWTs of treegion 1 and treegion 2 can be computed in turn as:

$$LBWT_{treegion1} = Max(LBET_{path\_A}, LBET_{path\_B}+LBWT_{treegion2});$$

$$LBWT_{treegion2} = Max(LBET_{paths\_in\_treegion2}).$$

For an outer loop body or a CFG containing loop structures, such as the CFG shown in Figure 2.6, the LBWT can be computed as follows,

$$LBWT_{treegion0} = Max(LBET_{path\_1}+LBWT_{loop\_A}, LBET_{path\_2}+LBWT_{treegion1}).$$

where LBWT of loop A is computed as $LBWT_{loop\_body\_A} * loop\_count\_A + LBWT_{treegion1}$. LBWT for the loop body ($LBWT_{loop\_body\_A}$) can be computed using Equation 2-4 if it contains more than one treegion and the loop count is determined from the workload specification or from profiling.

LBWT at the program level can be computed from the functional level LBWTs by traversing the function call graph using the leaf node first order. The LBWT of the '*main*' function represents the LBWT of the entire program.

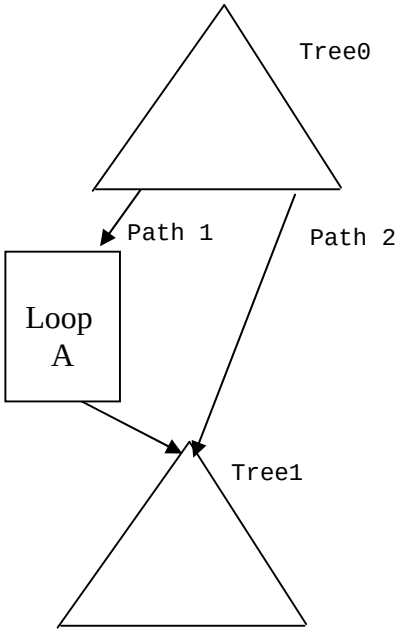


Figure 2.6. A CFG containing a loop structure.

As a final note, if we replace the LBET along each path with the actual schedule length/execution time, the LBWT becomes the WCET.

Next, we use a code example to illustrate the LBWT computation. The code example is a simple diamond structure as shown in Figure 2.7. Here, we use it to illustrate the LBWT computation and also show that treegion-based scheduling will result in a smaller LBWT compared to superblock scheduling or trace scheduling, making treegion scheduling more suitable for real-time applications.

First, we compute the superblock-based LBWT using the same 6-issue machine model as used in Section 2.3. The LBWT of the code segment is the same as $LBWT_{SB1}$, which is computed as follows using Equation 2-4:

$$LBWT_{SB1} = Max(LBET_{path\_1}+LBWT_{SB3},\ LBET_{path\_2}+LBWT_{SB2})$$

$$= Max(LBET_{path\_1}+LBWT_{SB3},\ LBET_{path\_2}+ LBET_{SB2\_path} +LBWT_{SB3})$$

$$= Max(8 + 1, 4 + 5 + 1) = 10 \text{ cycles.}$$

From this computation, it seems that control path 2 forms the critical path.

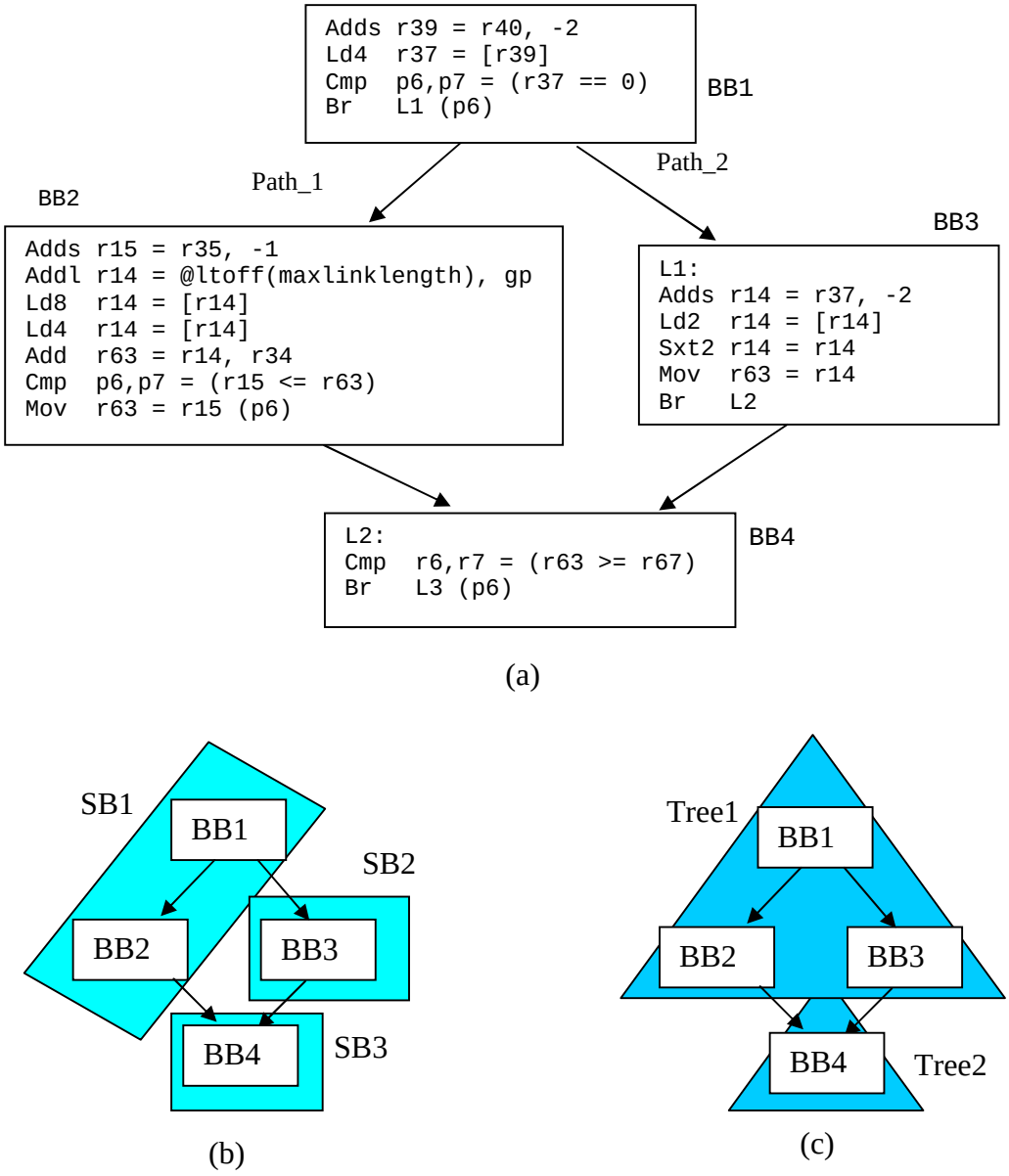


Figure 2.7. (a) The similar code example to Figure 2.3; (b) The superblocks formed without tail duplciation; (c) The natural treegion formed.

Using treegions as basic scheduling regions, the LBWT of the code segment is $LBWT_{tree1}$, which is computed as follows using Equation 2-4:

$$LBWT_{tree1} = Max(LBET_{path\_1}+LBWT_{tree2}, LBET_{path\_2}+LBWT_{tree2})$$

$$= Max(8 + 1, 5 + 1) = 9 \text{ cycles.}$$

The critical path is now due to control path 1. Compared to the superblock-based LBWT, treegions enable control speculation from BB3 along path 2. So, the execution time along path 2 is reduced to 6 cycles and the overall LBWT is reduced to 9 cycles. This illustrates that treegion scheduling is an appropriate scheduling framework for real-time applications due to its capability to perform speculation from multiple control paths simultaneously. Note that in this example, we do not consider the effect of branch prediction on WCET or LBWT for conciseness. We will include this in Chapter 4 when if-conversion optimizations are selectively performed.

Chapter 3 Compiling for Code Size Efficiency

In this chapter, we describe how we use our proposed profile-guided performance bounds to guide code compilation for code size efficiency. The objective is to selectively perform ILP optimizations so that significant ILP improvement is achieved at a very small cost in static code size increase. A brief background on ILP optimizations is contained in Section 3.1. Section 3.2 presents performance bound driven code size efficiency. Section 3.3 contains our proposed algorithm to regulate code size related ILP optimizations. The optimal tradeoff between performance improvement and code size is defined in Section 3.4, and a simple heuristic is developed to achieve this optimum. Section 3.5 contains the experimental methodology and results. A summary of this chapter is provided in Section 3.6.

3.1 Background on Code Size Related ILP Optimizations

A great number of code transformation techniques have been proposed in the literature to improve program performance. As our target in this chapter is code size

related optimizations for integer workloads, we focus on the three most commonly used ILP optimizations: tail duplication, loop unrolling and if-conversion.

Tail duplication (or code replication) replicates a subgraph of the control flow to remove side entries of a trace [6],[31] and to avoid conditional / unconditional branches [56]. Many instruction-scheduling approaches [28],[31],[51] use tail duplication in forming scheduling regions. Due to its evident impact on static code size increase, different heuristics have been proposed to decide whether a particular instance of tail duplication should be performed. One simple example is a threshold on the profiled execution frequency [31]. However, there is no systematic way to analyze the tradeoff between the cost in code size and the performance gain.

Loop unrolling is another technique used to enlarge a scheduling region. Modulo scheduling [59] may also benefit from loop unrolling to reach a non-integer MII [38]. However, it has been recognized that loop unrolling can degrade performance if it is not used judiciously due to increased code size and increased resource requirements. Sarkar [63] proposed a mechanism to automatically select an unroll vector for nested loops. His approach associates a cost model for feasible unroll vectors and the one with the best objective function is selected. The cost model evaluates an unroll vector without performing the unrolling. A similar approach to MII (RecMII and ResMII) computation is used as in modulo scheduling to estimate the ILP for a candidate unroll vector. In [36], an iterative compilation approach is proposed to search for the best unroll factor and tile size. Instead of a cost model, the actual execution time on the target machine is measured. While these approaches are mainly targeted at scientific codes, our focus is irregular integer workloads.

If-conversion [2],[58] replaces conditional branches with appropriate predicate computations, and the instructions that are control dependent on the branch are guarded with these predicates. The removal of frequently mispredicted branches can yield large performance gains [50]. Also, if-conversion increases the spatial locality of instructions and may reduce code size if the targeted instruction set architecture (ISA) uses predicate computation for a conditional branch, such as IA-64 [32],[65] or HPL-PD [35]. As pointed out in [4], full if-conversion generally works for compiling numerical applications. For integer applications, selective if-conversion [4] is essential to achieve performance gains due to the potential hazards of if-conversion [15]. Hyperblock formation involves a complex heuristic to choose which paths to be included and then performs if-conversion on the selected basic blocks [51]. Profile based selective if-conversion [55] uses profile information to compute the performance gain of if-conversion based on weighted schedule estimates before and after predicating a hammock. The schedule estimates are based on local scheduling results. Compared to this estimate, our performance bound calculation is more accurate as it considers the potential effects pf speculation on each scheduling region.

Note that all these optimizations have been proven to be very effective. The purpose of this chapter is not to reiterate the importance of these optimizations. Instead, our objective is to introduce a systematic way of regulating these optimizations so that performance gains are maximized at minor cost in static code size increase.

3.2 Performance Bound Driven Code Size Efficiency

In this section, we first define the notion of code size efficiency (CSEF). Then, we use tail duplication, loop unrolling, and if-conversion to explain how to use performance bounds to calculate this efficiency.

3.2.1 Code size efficiency

The major objective of code size related optimizations is to improve instruction level parallelism (ILP). One direct measure of the effectiveness of such a transformation is the ratio of ILP improvement over the code size increase. Since code optimizations are performed at compile time, we use static instructions-per-cycle (IPC) to measure ILP improvement. The static IPC is computed as the ratio of the number of retired instructions (IC) over execution time (ET). Both IC and ET are derived from profile information. The speculated instructions resulting from instruction scheduling are not included in IC. Using the ratio of ILP improvement over code size increase as a quantitative measure (as stated, such a measure is intuitively appealing and we will show later in Section 3.4 that it is indeed a good measure), two formal definitions of code size efficiency for code transformations are proposed.

First, we define the efficiency for an instance of a code transformation, called the *instantaneous code size efficiency*, as shown in Equation 3-1:

$$Efficiency_{inst.} = \frac{IPC_{after\_individual\_application} - IPC_{before\_individual\_application}}{code\_size_{after\_individual\_application} - code\_size_{before\_individual\_application}}$$

Equation 3-1

In Equation 3-1, the term in the numerator represents the ILP improvement of a particular instance of a code optimization, and the term in the denominator represents the cost of such an optimization in terms of static code size. Using loop unrolling as an example, if we unroll a particular loop once, the instantaneous efficiency of such an unrolling is the performance gain divided by the size of the loop body. Since there could be many loops in a program, there is one such instantaneous efficiency associated with each of them.

The definition in Equation 3-1 measures the performance impact at the cost of unit code size increase for a single instance of a code transformation. It is also useful to have a quantitative measure when more than one optimization instance has been performed. For example, assume a program has three loops. One unroll heuristic picks all three of them to be unrolled once and another heuristic may unroll just one loop many times. A quantitative measure would be able to tell which heuristic performs better in balancing performance and code size. Such a measure is what we define as *average code size efficiency*, shown in Equation 3-2.

$$Efficiency_{average} = \frac{IPC_{candidate} - IPC_{original}}{code\_size_{candidate} - code\_size_{original}}$$

Equation 3-2

Similar to Equation 3-1, average efficiency measures performance gains in terms of ILP improvement at the cost of code size increase. The difference is that Equation 3-1 is used to evaluate an individual optimization instance while Equation 3-2 is used for the combined impact of many instances of the same or different optimizations. In fact,

average efficiency can be viewed as averaging the instantaneous efficiencies of each individual code optimization that has been performed.

Note that the IPC improvement in Equations 3-1 and 3-2 closely correlates to the execution time reduction. In fact, we may use the ratio of execution time reduction over code size change to approximate code size efficiency (the difference between this ratio and the formal efficiency definition is a near constant factor for a given program). This ratio is easy to understand and intuitively appealing as it basically tells how many cycles can be saved at the cost of one additional instruction.

3.2.2 Using performance bounds to calculate code size efficiency

As shown in Equations 3-1 and 3-2, the ILP improvement of code optimizations is measured using static IPC, which involves two terms, IC and ET. IC is computed using block and edge profile information and remains constant as further increase/decrease of instructions due to code transformations and instruction scheduling are not counted. ET, however, varies (hopefully decreases) as a result of code transformations. To calculate the actual ET reduction, scheduled code is needed, which implies we need to perform instruction scheduling to evaluate the actual impact of a transformation. As instruction scheduling is time consuming ($O(N^2)$), such an approach is not practical. As discussed in Section 3.1, in practice, various heuristics are used to estimate benefits instead of performing instruction scheduling. Our approach is to use profile-guided performance bounds to evaluate the effectiveness of an optimization by how much the bounds are reduced. As a result, code size efficiency can be approximated as:

$$Efficiency_{inst} \approx \frac{LBET_{before\_individual\_application} - LBET_{after\_individual\_application}}{code\_size_{after\_individual\_application} - code\_size_{before\_individual\_application}}$$

Equation 3-3

3.2.3 Examples of code size efficiency computation

First, we focus on code transformations resulting in ILP improvement as well as code size increase. Both tail duplication and loop unrolling are such optimizations. Using a code segment from the benchmark *twolf* as an example, shown in Figure 3.1, we explain how to compute the code size efficiency.

The code segment shown in Figure 3.1 has two basic blocks (BB1 and BB2), a loop back edge (edge B), and a merge point (edges C and D), exhibiting the possibility of applying both loop unrolling and tail duplication.

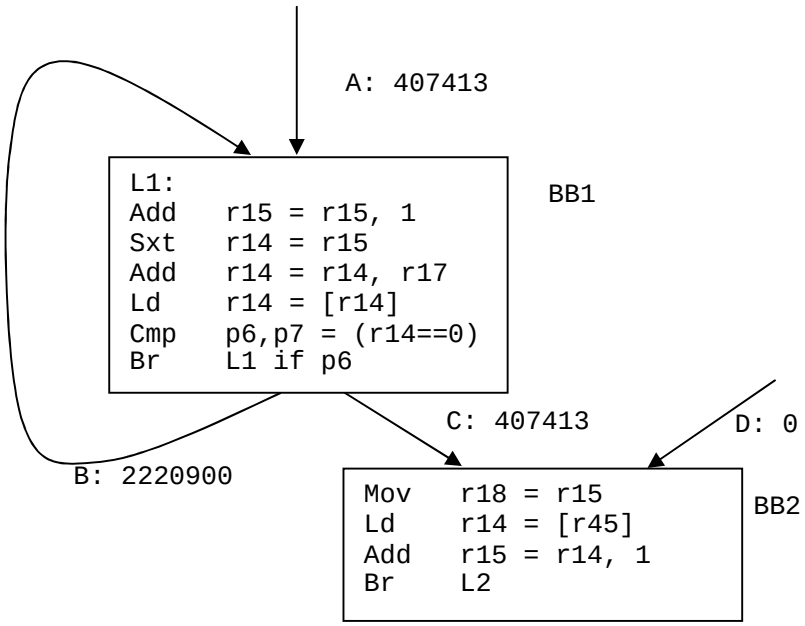


Figure 3.1. A code segment from *twolf* (in function new_dbox_a). Numbers along control edge labels are edge profiles.

Assuming load instructions have a 2-cycle latency and all other instructions in BB1 and BB2 have a 1-cycle latency (except CMP instructions which can be scheduled at the same cycle as the consuming branch), the lower bound execution time (LBET) before any transformation is the sum of the LBET of BB1 and the LBET of BB2. Assuming a 6-wide issue (2 ALU, 2 ALU/LD/ST, 2 ALU/BR) machine (which causes no resource constraints in this example), the LBET can be computed using Equation 2-1: LBET of BB1 is 6*2,628,313 = 15,769,878 cycles, LBET of BB2 is 3*407,413 = 1,222,239 cycles, and the sum is 16,992,117 cycles. After duplicating BB2, the instructions in BB2 can be scheduled in BB1 using control speculation, which results in an LBET of 15,769,878 cycles as the inclusion of BB2 instructions does not increase the true data dependence height (i.e., an LBET reduction of 1,222,239 cycles due to complete hiding of BB2 execution time). Therefore, the instantaneous code size efficiency of tail duplication occurring at the merge point of edges C and D is 1,222,239 / 4 = 305,560 cycle/instruction, i.e., one additional instruction leads to a 305,560 cycle execution time reduction.

Similarly, we can compute the efficiency of unrolling the loop body in Figure 3.1, i.e., BB1. As the loop-carried dependence height in this example is 1 cycle, the original loop body can overlap much of the computation with the unrolled copy. Here, we need to be careful in distributing profile data after loop unrolling. The probability propagation approach proposed by Wu and Larus [72] is used in this work, and the result of unrolling BB1 once is shown in Figure 3.2.

As shown in Figure 3.2, the probability propagation maintains the taken/not taken probability of the conditional branches at the end of BB1 and BB1' (the unrolled copy of BB1). After the profile is redistributed, the LBET of the loop body in Figure 3.2 (containing BB1 and BB1') can be computed using Equation 2-1 (9,751,148 cycles). Compared to the LBET of the loop body with no unrolling, LBET reduction is 15,769,878 – 9,751,148 = 6,018,730 cycles. Therefore, the instantaneous code size efficiency of loop unrolling (with factor 1) at back edge B is: 6,018,730 / 6 = 1,003,121 cycles/instruction.

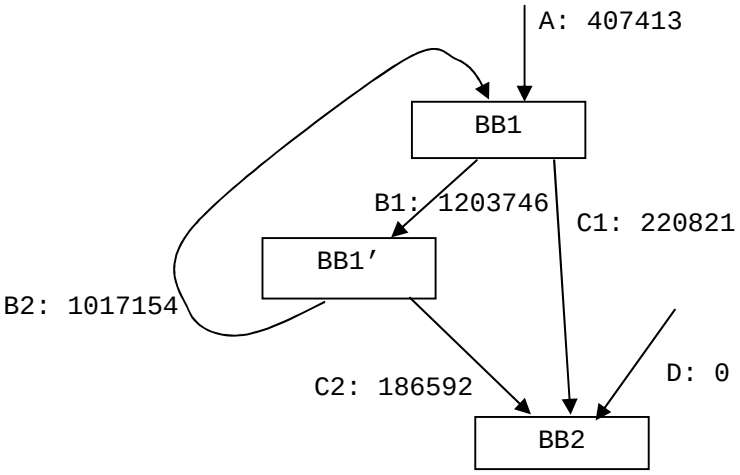


Figure 3.2. Loop unrolling of the loop body shown in Figure 3.1 with unroll factor of 2. (Numbers along control edge labels are edge profiles computed using probability propagation.)

If-conversion can *reduce* code size by removing branch instructions. Also, it may result in positive speedups by removing branch misprediction penalties. Therefore, the code size efficiency can be a negative number (i.e., positive speedup and negative code size increase), which represents one highly desired extreme of code size efficiency. (The other extreme of negative speedup and positive code size increase is what we always

want to avoid.) Using another simple code segment from the benchmark *twolf*, we show how we compute the efficiency of if-conversion by integrating branch misprediction penalties. The code segment is shown in Figure 3.3.

Using Equation 2-1, the LBET of the region containing BB1, BB2 and BB3 is computed as 28,111*2+169,174*3 = 563,744 cycles. Then, we consider potential branch misprediction penalties. Assuming static branch prediction and a 10-cycle misprediction penalty for each misprediction, the overall misprediction penalty of the conditional branch in BB1 is 28,111*10 = 281,110 cycles. If the profile of dynamic branch prediction is available, more accurate penalty computation can be used.

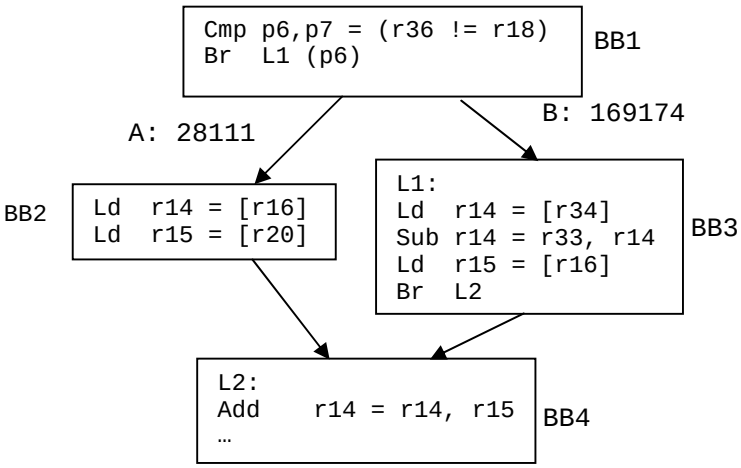


Figure 3.3. A code segment from *twolf* (function add_penal) to show efficiency of if-conversion. Numbers along control edge labels are edge profiles.

After if-conversion, the branches in BB1 and BB3 are removed (i.e., 2-instruction reduction) and the resulting LBET is 3*(28,111+169,174) = 591,855 cycles, which means a reduction of (563,744+281,110-591,855) = 252,999 cycles. Note that this computation involves only the control dependent blocks of the conditional branch (BB1, BB2 and

BB3). It does not depend on the merge block (BB4), and the same result holds when BB4 has more than 2 entry edges.

As pointed out previously, optimizations with positive speedups and negative code size increase are always performed. So, we do not need to calculate the actual efficiency for such cases. For if-conversions that have both negative speedup and negative code size increase, positive code size efficiency results. Such efficiency implies that we may want to perform if-conversion with low positive efficiency to reduce code size (although hurting performance slightly) and use the saved code size for optimizations with higher efficiency.

3.3 Regulating Code Size Related ILP Optimizations

Based on the quantitative measure of code size efficiency defined in Section 3.2, we develop an algorithm to regulate code size related optimizations, as shown in Figure 3.4.

The algorithm in Figure 3.4 has three steps in regulating different kinds of code size related transformations. As a preparation step, we construct basic scheduling regions without performing any region-enlarging optimizations. The examples are treegion formation without tail duplication (i.e., the natural treegion) and superblock formation without tail duplication. Such regions are single-entry multiple-exit regions for which LBET can be computed using Equation 2-1.

Algorithm for regulating code size related optimizations

0. Form basic scheduling regions to facilitate LBET computation and to identify program structures that are candidates for optimizations.
1. Perform code size reducing optimizations: if-conversion
    a. For a diamond / hammock structure, compute performance gains of if-conversion.
    b. If the if-conversion produces positive (or zero) LBET reduction, perform it.
    c. If the performed if-conversion results in a new diamond / hammock for its parent branch, continue to check this parent branch for if-conversion.
    d. Repeat step 1a – 1c, until no more diamond / hammock structures need to be checked.
2. Perform code size increasing optimizations: loop unrolling and tail duplication
    a. Compute instantaneous code size efficiency for each loop unrolling / tail duplication candidate using Equation 3-1.
    b. Search the candidate list to find the one with the best efficiency.
    c. If the selected candidate passes the feasibility check, perform the optimization and update the efficiency of candidates affected by the optimization. The feasibility check may involve code size constraints, register pressure, etc..
    d. Repeat step 2a – 2c, until the overall code size reaches a limit or there are no more candidates.

Figure 3.4. The algorithm for regulating code size related optimizations.

Optimizations are treated differently based on their code size efficiency characteristics. Optimizations with positive speedup and negative code size increase are examined first in Step 1 of the algorithm. Then, an iterative approach is used to selectively perform code-expanding optimizations, as shown in Step 2 of the algorithm. First, step 2a computes the efficiency of all potential optimization instances. Then, the best candidate is found from these instances based on their efficiency in step 2b. Next, if the one with the best efficiency passes the feasibility check, it will be performed in step

2c. The feasibility check basically makes sure that a particular optimization will not result in excessive resource utilization, e.g., the size of a loop body is less than the level one I-cache size. As one particular optimization may change the efficiency of another optimization or enable another optimization (e.g., a tail duplication may enable a diamond/hammock to be constructed for if-conversion), a local efficiency update is performed in Step 2c if one optimization instance is performed. Note that this iterative approach can automatically choose a good unroll factor for a loop by unrolling the original loop body one iteration at a time.

3.4 Optimal Tradeoff between ILP Improvement and Code Size Increase

For code size increasing optimizations, the algorithm in Section 3.3 iteratively selects and performs those with the best code size efficiencies. If we use a curve to represent the resulting ILP improvement and relative code size increase, which we call the *ILP vs. code size curve*, a very interesting phenomenon is revealed: optimizations among initial selections exhibit large performance improvement with small code size increase (i.e., high efficiency) and those selected later on show quickly dropping performance improvement with relatively larger code size increase (i.e., low efficiency). Such a phenomenon exhibits the effect of '*diminishing returns*', as we can see from Figure 3.5.

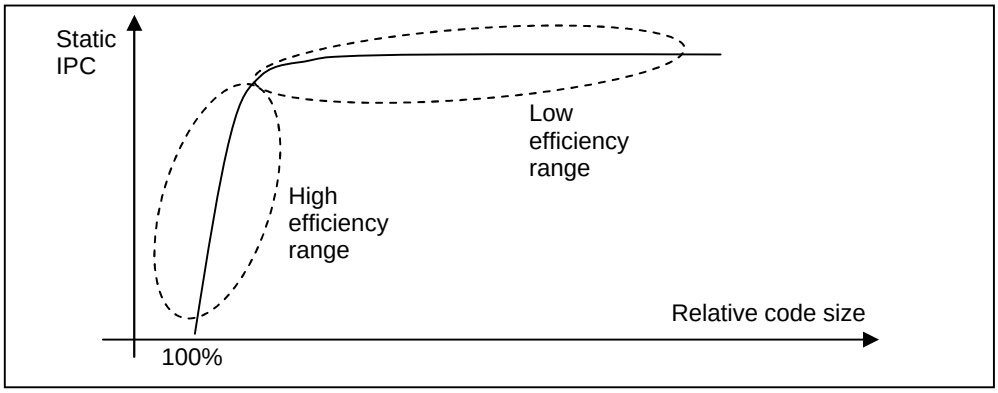


Figure 3.5. An example curve showing the relationship of ILP improvement and code size increase.

Figure 3.5 shows an example ILP vs. code size curve, which exhibits common characteristics of individual benchmarks we studied (see Section 3.5.3). The diminishing returns are due to the rapidly decreasing code size efficiencies, which in turn is due to the following two fundamental reasons. First, based on the definition of code size efficiency, an instance optimization with high efficiency should have high execution frequency. The well-known '*90/10 rule*' points out that a small part of the static code (hot portions) consumes most of the execution time. After performing optimizations in these hot portions of code, the remaining optimizations should have much lower efficiencies due to the much lower execution frequency. Secondly, high efficiency also requires that the resulting code must have better performance bounds, i.e., the instance optimization must reduce the DDG height without causing any resource conflict problems. This requirement filters the optimizations applied in hot portions of a program.

The diminishing returns phenomenon shown in Figure 3.5 enables us to define the optimal tradeoff between ILP improvement and code size increase. One natural choice is

the ‘knee’ of the curve in Figure 3.5, provided that the corresponding code size still satisfies the overall feasibility check.

To automatically find this knee in the curve, a simple heuristic is developed by taking advantage of the steep slope of the high efficiency part of the ILP vs. code size curve, as shown in Figure 3.6. Figure 3.6 replicates the ILP vs. code size curve in Figure 3.5 and the knee of the curve is marked as point *A*.

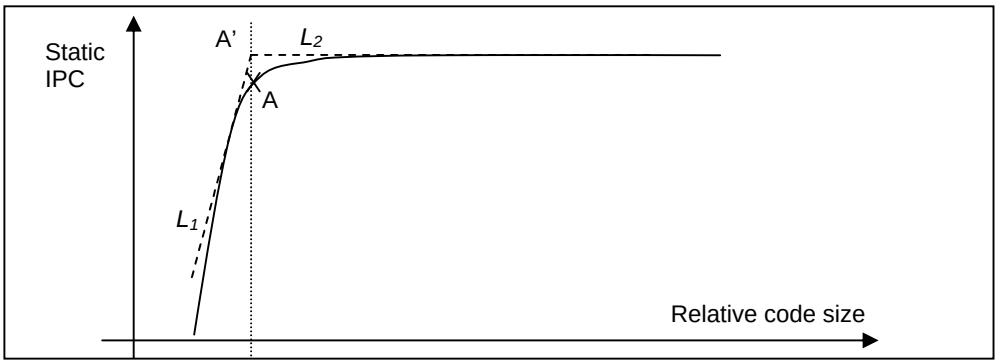


Figure 3.6. Achieving the optimal tradeoff between ILP improvement and code size increase.

To locate *A*, we can first use two straight lines to approximate the curve (as the dashed lines L_1 and L_2 shown in Figure 3.6). Then, the knee of the curve becomes the intersection, *A*’, of these two lines. A simple threshold scheme can be used to find *A*’: the point along the curve whose slope is between the slope of L_1 and the slope of L_2. The slope of the ILP vs. code size curve represents the ratio of static IPC changes over relative code size changes, which is exactly the definition of the instantaneous code size efficiency in Equation 1. So, the approach to achieve the optimum tradeoff is simply as follows: *perform the optimizations whose instantaneous code size efficiency is higher than the threshold efficiency K*. This threshold efficiency can be any value between the slope of L_1 and the slope of L_2. In other words, the range between the slope of L_1 and

slope of L_2 determines the *robustness* of this threshold scheme. In our experiments (see Section 3.5.4), we vary K from $\tan(\pi/12)$ (corresponding to a line with an angle of 15 degrees) to $\tan(\pi/6)$ (corresponding to a line with an angle of 30 degrees) to show the robustness of this threshold scheme. This threshold K is both *workload-independent* and *input-independent.*

As we use the ratio of LBET change over absolute code size increase (measured in number of instructions) to compute code size efficiency, we can further derive the threshold scheme as in Equation 3-4. The derivation details can be found in Appendix A.

$$\frac{d(-LBET)}{dSize_{absolute}} \geq \frac{K * LBET}{IPC_{static} * IC_{static}} \qquad \textbf{Equation 3-4}$$

In Equation 3-4, IC_{static} represents the static operation count of the program (i.e., the static program size; whereas the term $IC_{dynamic}$ is the number of retired instructions during execution and is used for IPC calculation), K is the threshold on instantaneous code size efficiency, *LBET* is the lower bound of execution time for the whole program, *d*(-*LBET*) is the reduction in the lower bound (both are computed using Equation 2-1), and IPC_{static} (= $LBET$ / $IC_{dynamic}$) represents the ILP feature of the original program.

3.5 Experimental Results

In this section, we first describe our experimental methodology and present results using the algorithm in Section 3.3. Then, we show the effectiveness and robustness of the threshold approach in Section 3.4.

3.5.1 Methodology

In our experiments, we use the SPEC CINT 2000 benchmarks [30] to evaluate the proposed algorithms. The benchmarks are first compiled into IA-64 assembly using the *gcc* compiler (version 3.1). As our purpose is to regulate ILP optimizations, we use the level one optimization provided by *gcc* to perform classical optimizations (as discussed in Section 3.5.2, a by-product of the level one optimization is that *gcc* produces predicated code). The resulting IA-64 assembly codes are then parsed into the LEGO compiler framework [41], which we use to implement the algorithms in this chapter. The IA-64 assembly is instrumented and executed to gather profile information. In our experiments, we use the reference input data set and skip the first 500 million instructions and profile the next 500 million instructions for each benchmark.

Treegion-based instruction scheduling is used in the LEGO compiler. We first use natural treegions (formed without any tail duplication) to get the baseline execution time and static IPC. Performance bounds calculated using Equation 2-1 are used as the baseline execution time, which represents the best schedule achievable without any further optimization. The baseline results are show in Table 3.1, which includes static code size, the number of dynamic retired instructions (around 500M as we profiled 500M instructions) and the lower bound of execution time. Static IPC indicates the inherent ILP present in the current code and the results show that many benchmarks have moderate ILP (IPC around 2) while the benchmark *gap* has very limited ILP (IPC around 1). An examination of the benchmark *gap* finds that the function *ProdInt* is heavily executed in our profile phase. The complex computations (long dependence chain) in this function result in low ILP.

In Table 3.1, we also include the ratio of estimated execution time of treegion-scheduled code over the lower bound. The execution time of treegion-scheduled code is computed using a scoreboard dependency-enforcing approach (i.e., it is the execution time assuming ideal caches and ideal branch prediction). From these results, it can be seen that the treegion scheduler produces quite a good schedule, exceeding 1% to 13% of the lower bound. The mismatch is because the performance bound is calculated assuming that all false register dependencies can be removed by software renaming, and that control dependencies can be minimized by multiway branch transformations. Such assumptions are too optimistic as liveness beyond the basic block scope may require a copy instruction to be inserted. Resource conflicts due to speculation from multiple paths in a treegion are another reason.

Table 3.1. Baseline results including static code size, execution time, and static IPC.

Baseline	bzip	crafty	gap	gzip	mcf	parser	twolf	vortex	Vpr
Static size (num of insn.)	7543	51085	131447	13316	2548	25545	65786	120735	35416
Number of dynamic insn. Retired	498M	490M	500M	495M	491M	496M	496M	499M	497M
Lower bound of exe. time (cycles)	257M	217M	495M	275M	276M	263M	325M	219M	318M
Static IPC	1.93	2.26	1.01	1.80	1.78	1.87	1.53	2.27	1.56
Ratio of natural tree schedule results over the lower bound	104%	108%	104%	112%	106%	113%	107%	107%	101%

3.5.2 Regulating code size decreasing optimizations – if-conversion

Step 1 of the algorithm shown in Figure 3.4 regulates how code size decreasing optimizations, if-conversion in this chapter, are performed. Due to its code size reduction effect, any if-conversion, which produces positive (or zero) LBET reduction (i.e., positive speedups), will always be performed. As described in Section 3.3, we use static branch

prediction to estimate branch misprediction penalties assuming that each misprediction incurs a 10-cycle penalty.

Table 3.2 shows the if-conversion results using our algorithm. As stated previously, the input IA-64 assembly code is generated using the GNU *gcc* compiler with level one optimizations, which perform not only classical optimizations but also if-conversion. By applying our algorithm to this already if-converted code, we show that our algorithm can improve upon *gcc*'s if-conversion algorithm.

Table 3.2. If-conversion results.

	bzip2	crafty	gap	gzip	mcf	parser	twolf	vortex	vpr
If-conversions (by gcc)	113	780	2852	139	61	502	1042	1692	325
Number of conditional br.	487	2712	9747	819	167	2068	3625	7469	1805
If-conversion with pos. gain	2	40	1	6	5	2	11	4	10
If-conversion with zero gain	19	163	324	26	7	80	445	191	74
If-conversion with neg. gain	4	58	2	3	4	1	24	37	8
No if-conversion: complex CFG	358	1608	5752	546	133	1483	2787	2161	1263
No if-conversion: ret_call	104	843	3668	238	18	502	358	5076	450
Number of dynamic cond. br.	36.4M	23.8M	23.6M	37.4M	71.0M	46.0M	33.8M	32.8M	30.5M
Reduction in execution time including br. misprediction penalty (cycles)	91016	1057363	9700	799877	90688	80125	506372	122592	13148695
static br. misprediction rate	7.34%	12.78%	11.61%	10.42%	15.44%	12.03%	13.40%	0.92%	14.35%

Interesting observations can be made from Table 3.2. The first row in Table 3.2 reveals that *gcc* has removed a significant amount of conditional branches through predication, although the second row, which shows the number of existing conditional branches in each benchmark after *gcc*'s if-conversion, suggests that there still exist potential if-conversion candidates. Our algorithm examines those conditional branches and confirms that the majority of these conditional branches are hard to if-convert. We report those hard-to-convert conditional branches in two categories: row 6 shows the number of conditional branches followed by a complex CFG (e.g., merging points at both if path and else path of a diamond/hammock) inhibiting diamond/hammock detection, and row 7 presents the number of detected diamonds/hammocks containing function call, return, or indirect branch instructions. (We excluded the case where both paths contain the same function call or return instruction) In such cases, if-conversion may hurt branch prediction performance as it may introduce more conditional function calls and returns, which in turn incur branch misprediction penalties. For those if-convertible branches, our algorithm computes the performance gain. Using the benchmark *gzip* as an example, *gcc* converts 139 conditional branches and there remain 819 conditional branches in the program. Our algorithm finds that 546 of them do not form a diamond/hammock structure. For those that form a diamond/hammock, 238 of them have at least a function call or a return along one or both paths. For the remaining ones, 6, 26, and 3 of them produce positive, zero, and negative speedups, respectively.

Next, we analyze the performance impact of if-conversion. In this experiment, we perform only the if-conversion instances that produce positive gains. Although the

number of these if-conversion instances seems limited (1 to 40), significant performance gains can be achieved, as shown in Figure 3.7.

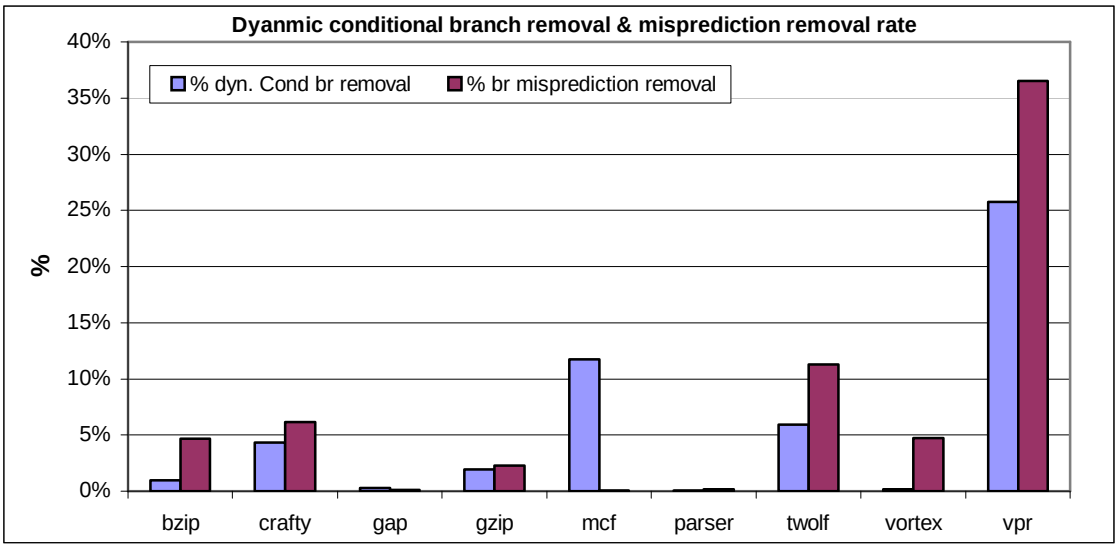


Figure 3.7. The removal rate of dynamic conditional branches and mispredictions by if-conversion.

Figure 3.7 shows the percentage of dynamic conditional branches and associated mispredictions removed by if-conversion. It can be seen that 0.1% (*parser*) to 25.8% (*vpr*) of dynamic conditional branches can be eliminated, and 0.1% to 36.6% of branch mispredictions associated with these conditional branches can be removed, assuming static branch prediction. Note that a higher rate of dynamic branch removal does not necessarily mean a higher reduction in mispredictions. For example, if-converting 5 conditional branches in the benchmark *mcf* reduces the number of dynamic conditional branches by 12%, which only results in 0.1% reduction in branch mispredictions. The reason is that the removed conditional branches are highly biased, which in turn produces a small reduction in LBET as shown in row 9 of Table 2. For completeness, the number of dynamic conditional branches is shown in row 8 and static branch misprediction rates for conditional branches are included in the last row of Table 3.2.

Finally, we analyze the code size reduction impact of if-conversion. We choose to perform if-conversion instances with positive or zero gains in this experiment. Assuming each conversion saves two instructions in IA-64 assembly, the overall code size reduction is computed and is shown in Figure 3.8. Remember that this reduction is achieved on the IA-64 code that has already been predicated by *gcc*. This demonstrates that our algorithm reduces code size by performing if-conversion more aggressively. From Figure 3.8, it can be seen that if-conversion reduces static code size consistently for every benchmark, up to 1.4% (the benchmark *twolf*) and 0.68% on average. Although these numbers seem to be trivial, in the next subsection, we will show that utilizing such a small amount of code size can lead to very large ILP improvements.

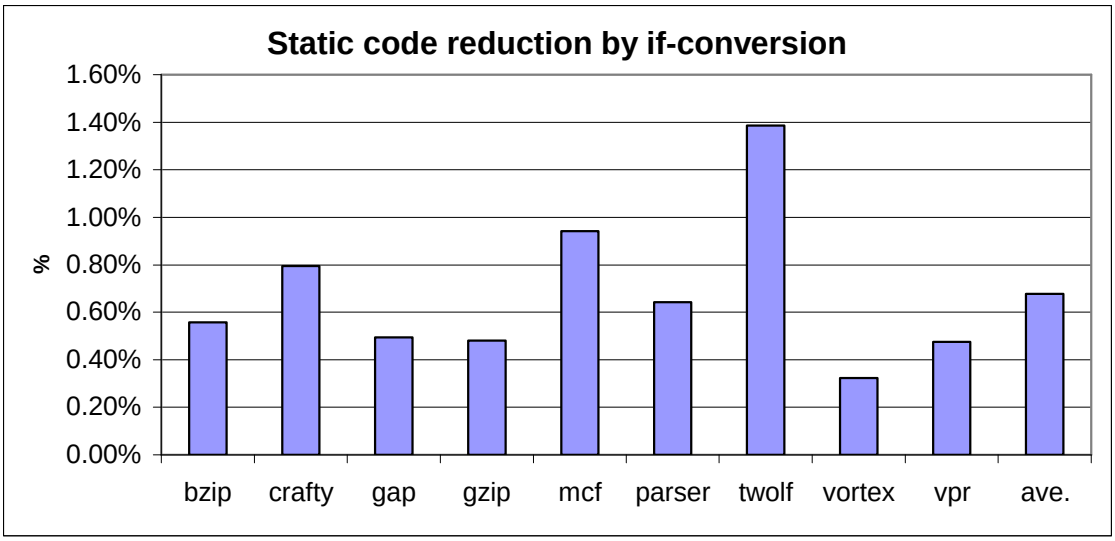


Figure 3.8. The static code size reduction by if-conversion.

3.5.3 Results of regulating code size increasing optimizations – tail duplication and loop unrolling

Step 2 of the algorithm shown in Figure 3.4 regulates code size increasing optimizations (tail duplications and loop unrolling). It iteratively selects and performs the one instance of tail duplication or loop unrolling with the highest instantaneous efficiency. In this experiment, we examine the effectiveness of such an iterative approach. For each benchmark, we set the limit of overall code size increase at 1%, 2%, and 5% of its original size (i.e., the optimization stops when the overall code size reaches this limit). The corresponding ILP improvements are shown in Figure 3.9.

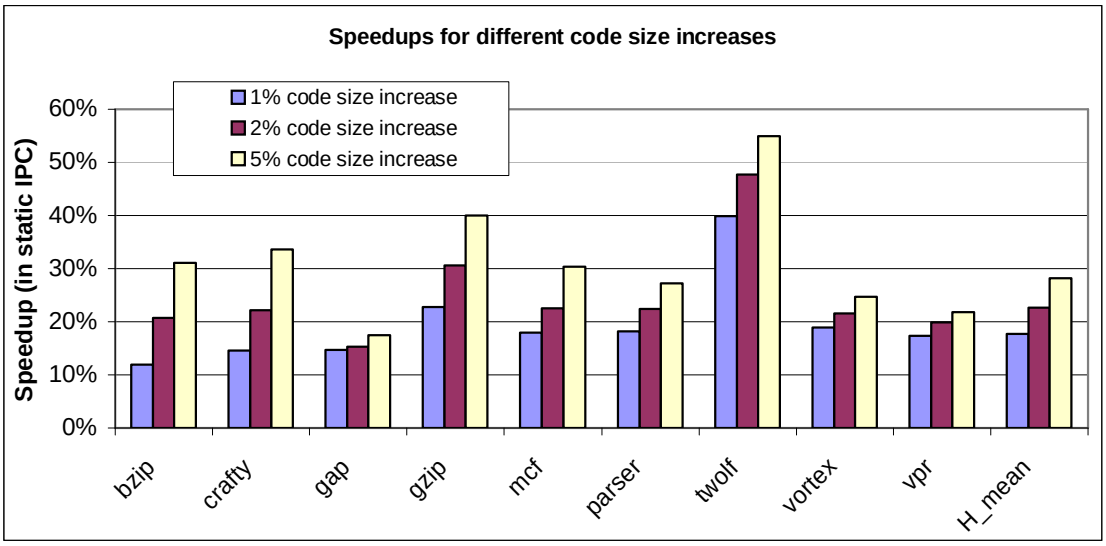
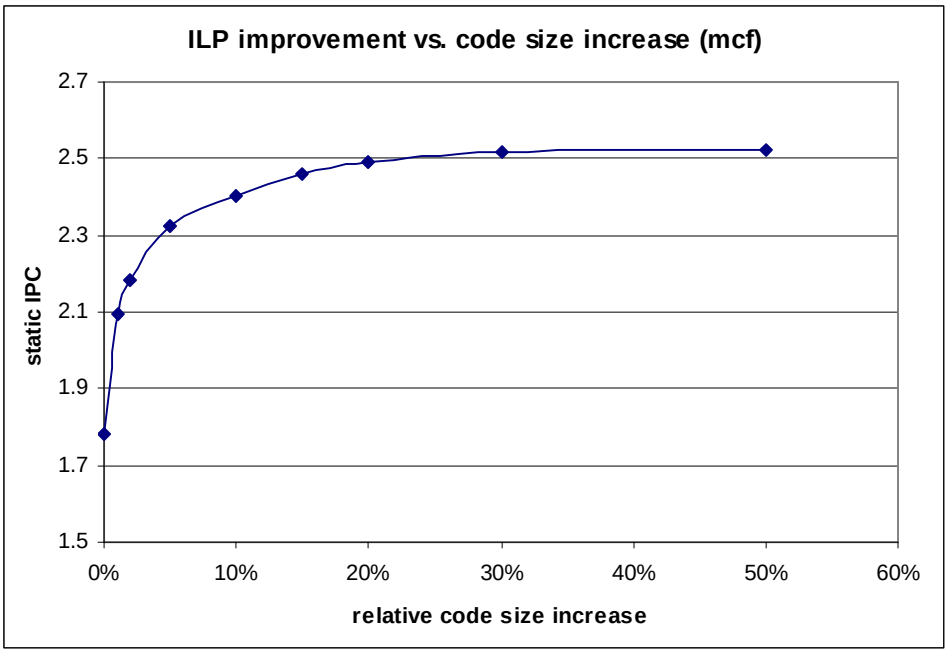


Figure 3.9. The speedups for different code size increases.

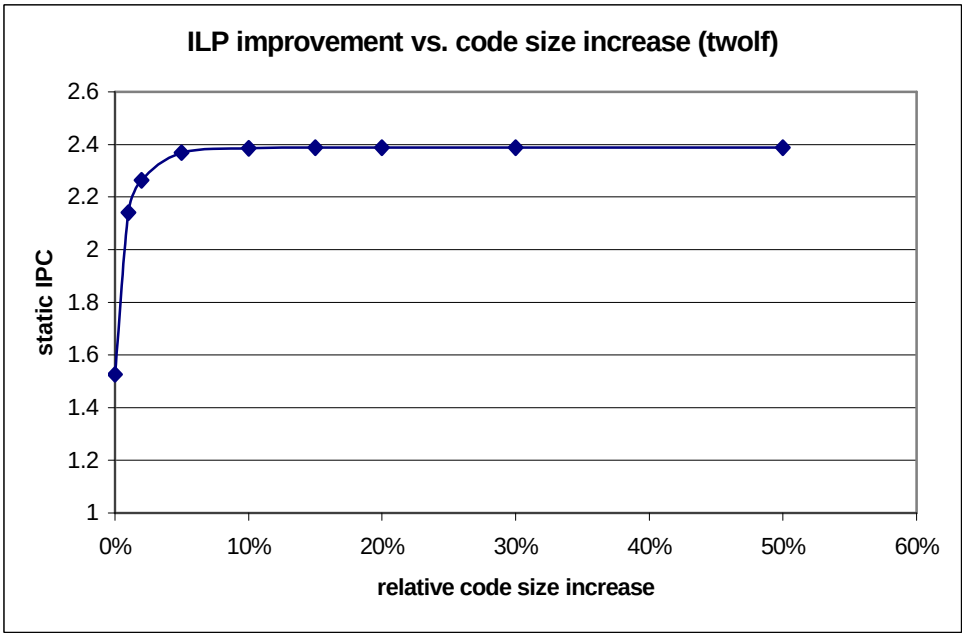
Two major observations are made from Figure 3.9. First, it is evident that a very small amount of code-size increase can lead to significant improvement in ILP if this code size is used judiciously. Our algorithm achieves up to a 40% increase and an average 18% increase in static IPC when the code size is expanded by just 1%. This, on the other hand, emphasizes the importance of code size reducing optimizations: the code

size saved by aggressively performing code-reducing transformations can be used for code expanding optimizations with high efficiency. This is the reason that the algorithm in Figure 3.4 performs code-reducing transformations before code enlarging ones. Secondly, it can be seen from Figure 3.9 that further code-size increase has less impact on ILP improvement. As shown in the figure, an additional speedup of 5% on average is observed as the code size increases from 1% to 2% of its original size, still significant but less impressive compared to 18% for the first 1% of code size increase. The reason is that during the iterative selection process, the efficiency of the selected optimization decreases rapidly. Using the benchmark *vortex* as an example, the first selected optimization is one tail-duplication in procedure *Chunk_ChkGetChunk* with an efficiency as high as 534,609 cycles/instruction. After another 7 optimizations were selected and performed (resulting in replicating 66 instructions), the efficiency of the next chosen optimization drops to 77,484 cycles/instruction. As discussed in Section 3.4, two main reasons account for such 'diminishing returns': the '90/10' rule and the reduction in data dependence height without causing resource conflicts.

Next, two individual benchmarks, *mcf* and *twolf*, are chosen as representative benchmarks to examine the impact of diminishing returns in detail. A curve of ILP vs. code size is shown for each benchmark in Figure 3.10.



(a)



(b)

Figure 3.10. ILP improvement vs. code size increase for benchmarks (a) *mcf* and (b) *twolf*.

In Figure 3.10, the code size increase of each benchmark is normalized to its original code size. The curve of ILP improvement vs. code size increase is obtained as follows. First, we set the limit of relative code size increase to 1%, 2%, 5%, 10%, 15%,

20%, 30%, and 50% and use the iterative approach to selectively perform code size increasing optimizations. Then, we produce the curve by interpolating these results. From these two benchmarks, we can see that diminishing returns usually happen quickly with small code increase. For the benchmark *twolf*, it happens at approximately 5% code size increase while the benchmark *mcf* shows that the performance can still be improved significantly until the increase is approximately 20%.

3.5.4 Achieving the optimal tradeoff between ILP improvement and code size increase

As discussed in Section 3.4, the diminishing returns that are observed in Figure 3.10 enable us to define the optimum tradeoff between ILP improvement and code size increase. Also, a threshold scheme is developed to achieve this optimum. In this experiment, we show the effectiveness as well as the robustness of this threshold scheme.

First, we examine the robustness of our scheme. We set K to $\tan(\pi/6)$ and compute the threshold on instantaneous code size efficiency for each benchmark using Equation 3-4, as shown in Table 3.3. Based on the threshold values, the optimizations whose efficiency exceeds the threshold are performed. The resulting code size and ILP improvement are also shown in Table 3.3. It can be seen that, for many benchmarks, the resulting optimal tradeoff has a small code size increase (up to 18%) and a very large ILP improvement (up to 59%).

Table 3.3. The resulting code size and ILP improvements when threshold *K* = 0.577.

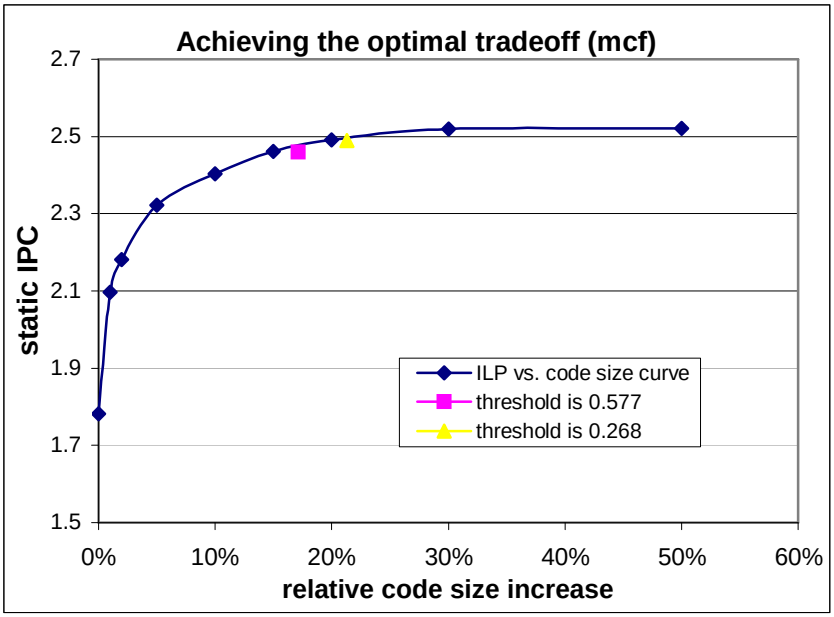
	bzip	crafty	gap	gzip	mcf	parser	twolf	vortex	vpr
Efficiency threshold (cycles/instruction)	10211	1088	2153	6594	35022	3167	1867	461	3307
Resulting relative code size increase	18.2%	9.7%	1.5%	11.2%	17.1%	13.5%	4.8%	5.3%	5.5%
Resulting static IPC increase	59.4%	39.6%	15.0%	48.6%	38.0%	34.1%	55.9%	24.4%	33.1%

Then, we change *K* to $\tan(\pi/12)$ and re-calculate the threshold values, as shown in Table 3.4. Compared to Table 3.3, it can be seen that the rather large change in threshold value (over 100%) results in very small variations in ILP improvement (up to 2.4%) and code size (up to 4.2%). This demonstrates the robustness of our threshold scheme.

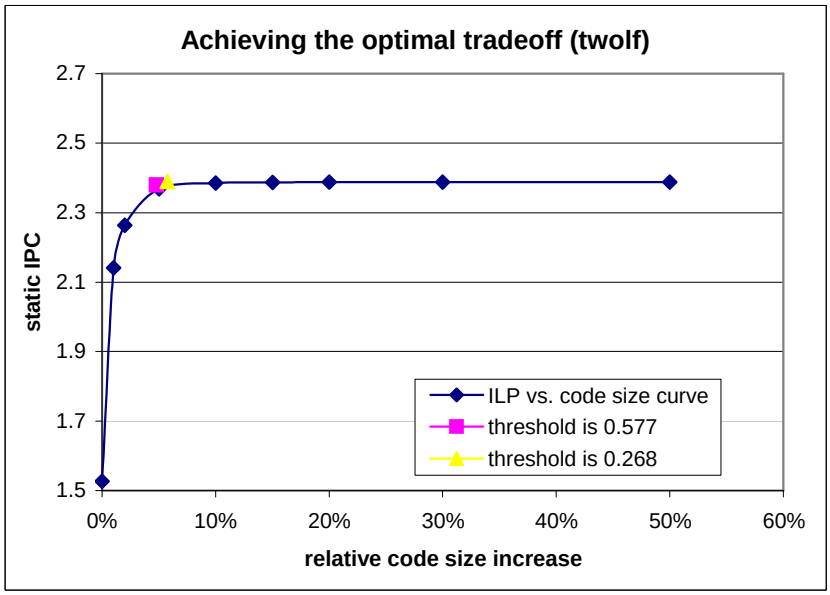
Table 3.4. The resulting code size and ILP improvements when threshold *K* = 0.268.

	bzip	crafty	gap	gzip	mcf	parser	twolf	vortex	Vpr
Efficiency threshold (cycles/instruction)	4743	505	1000	3063	16267	1471	867	214	1536
Resulting relative code size increase	21.4%	13.8%	2.2%	13.4%	21.3%	18.6%	5.8%	7.5%	6.7%
Resulting static IPC increase	60.9%	40.9%	17.4%	49.7%	39.7%	35.7%	56.6%	24.9%	33.3%

Next, we use the representative benchmarks used in Section 3.5.3, i.e., the benchmarks *mcf* and *twolf* to show that our scheme does achieve the optimal tradeoff (i.e., the knee of the ILP vs. code size curve). Note that *mcf* shows the maximal variation in resulting code size for different thresholds, representing the worst case among all these benchmarks. The results are shown in Figure 3.11.



(a)



(b)

Figure 3.11. Achieving the optimal tradeoff between ILP improvement and code size increase. (a) benchmark *mcf*, (b) benchmark *twolf*.

Figure 3.11 shows the tradeoff points obtained by our threshold scheme with different threshold values as well as the ILP vs. code size curve. The curve for the benchmark *twolf* shows a sharp turn around the knee, and our algorithm finds the optimal tradeoff (or knee of the curve) precisely. For the benchmark *mcf*, the ILP vs. code size

curve exhibits a less sharp turn around the knee. As a result, our algorithm generates two more distinct points along the curve. However, it can be seen that both points are still close to the ‘knee’ and both of the points are efficient solutions.

3.6 Summary

Based on a bound-driven notion of code size efficiency, a novel approach is developed in this chapter to regulate code size related ILP optimizations in a systematic way. Three types of commonly used ILP transformations: if-conversion, loop unrolling, and tail duplication are considered. Our algorithm examines code size reducing optimizations first. Then, an iterative approach is used to selectively perform code-enlarging optimizations with the best efficiency. In such a way, maximal ILP improvement can be achieved with minimal static code size increase. Experimental results using the SPEC CINT 2000 benchmarks show that a very high ILP improvement (up to 40% and 18% on average) can be achieved with a very small code size increase (1%). Considering the code size saved by if-conversion, the overall code size increase is further reduced (–0.4% to 0.7% overall increase).

In this chapter, we also show the interesting diminishing returns phenomenon in performing code-enlarging optimizations to improve ILP. The optimal tradeoff between the ILP improvement and code size increase can be defined as the knee of the ILP vs. code size curve. Then, a threshold scheme is developed to achieve this optimum. Experimental results demonstrate that our threshold scheme is effective and robust in achieving the optimal tradeoff.

Chapter 4 Code Size Aware Compilation for Real Time Applications

In this chapter, we use profile-independent performance bounds to selectively perform code optimizations to reduce the worst-case execution-time (WCET) of real-time applications at a minor cost in static code size increase. First, we present a brief background summary in Section 4.1. In Section 4.2, we advocate using in-order VLIW/EPIC type microarchitectures to exploit ILP in real time applications. On the one hand, the compiler controlled plan of execution (POE) makes the worst-case execution-time (WCET) analysis more accurate as run-time variations are minimized. On the other hand, the compiler can leverage ILP optimizations and instruction scheduling to explicitly reduce the WCET of real-time tasks, which in turn improves the system level schedulability. We also show that treegion scheduling suits real-time systems well due to its ability to optimize multiple control paths simultaneously. Section 4.3 describes an extended measure of code size efficiency to account for the WCET reduction and code size increase. A similar algorithm regulating ILP optimizations to the one in Section 3.3

is presented in Section 4.4. The experimental methodology and results are given in Section 4.5. Section 4.6 summarizes this chapter.

4.1 Background

In real-time systems, a task needs to satisfy both functional and temporal requirements to achieve overall correctness [22],[45]. The functional requirements are defined based on program semantics to generate correct outputs from inputs and the temporal requirements define the upper bounds (or deadlines) for such input-output transformations. A real time system may have many such tasks (periodic or sporadic), which are scheduled (called task scheduling) to meet the overall requirements of the system.

In order to guarantee a task to be finished by a specified deadline, worst-case execution-time (WCET) analysis is commonly used. Task scheduling then sets different priorities for different tasks accordingly. Due to its evident impact, task scheduling for real-time applications is an active research topic [23],[45]. In this dissertation, we look at the problem from a different point of view. Instead of focusing on task scheduling algorithms, we focus on intra-task, instruction-level scheduling. More specifically, we use performance bounds to guide *code optimizations* and *instruction scheduling* so that the WCET of each task is reduced, which in turn increases the schedulability of the whole system.

Most of the previous work on real-time scheduling takes a compiled program as input and performs either static or dynamic timing analysis to determine the WCET. Little work is done at the instruction level (optimization or scheduling) to explicitly

reduce the WCET. Gerber and Hong [22] proposed a scheduling approach called structural code motion based on trace scheduling. First, the task-level timing requirements are broken down into the event level. A new language based on the *C* language, called the Time-Constrained Event Language (TCEL), is developed to express detailed event-based timing constraints. Then, the code is partitioned into sections based on observable events. Trace-based scheduling is used to schedule each section. The critical traces (the traces with execution times larger than the timing constraints) are examined and code motion may be performed to move operations across sections so that the WCET relationship between observable events satisfies the timing constraints. Such code motion can be unconditional (or safe) or control speculative. Since trace scheduling focuses on one trace at a time, such code motion results in considerable bookkeeping code and could potentially increase the criticality of other traces. As a result, the critical paths are repetitively checked and scheduled. Compared to this approach, treegion based scheduling enables speculation from multiple control paths simultaneously and limits the enumeration of critical paths. Another technique to increase task schedulability is based on the concept of imprecise computation [26],[46]. If the purpose of some computation is known statically as refining the results, such computation can be skipped without affecting system sustainability, i.e., the quality of computation is traded for the timeliness of the results.

In [42],[70], algorithms have been proposed to schedule instructions with timing constraints (release times and deadlines) on ILP processors. These algorithms are targeted toward single-issue pipelined processors and work on the basic-block level. The proposed algorithms guarantee to find a feasible schedule for a range of special cases.

4.2 Explicitly Parallel Instruction Computing (EPIC) in Real-Time Systems

In this section, we advocate using explicitly parallel instruction computing (EPIC) architectures for real time systems. On one hand, EPIC architectures exploit instruction-level-parallelism (ILP) aggressively to achieve high performance at a reduced level of hardware complexity [65]. On the other hand, the design philosophy of EPIC puts the software/compiler in total control of dynamic execution and how ILP is exploited. In EPIC architectures, the latency of each operation is exposed to the compiler, and the plan of execution (POE) [65], including when an instruction is to be executed and which function unit is to be used, is specified by the compiler. Such features suit the purpose of real-time systems well as they facilitate accurate static WCET analysis and easily integrate the WCET analysis with code optimizations and instruction scheduling to focus on reducing WCET.

Treegion-based global scheduling [28],[78] aims for high performance for wide issue VLIW / EPIC processors although it can be applied to superscalar processors as well. In addition to providing a large scheduling scope, i.e., a treegion, it has the ability to speedup multiple control paths in a treegion, thus making it more suitable for real-time applications than trace scheduling or superblock scheduling as the speculation impacts on multiple paths are considered simultaneously. Also, treegions do not have side-entries, thereby avoiding the overhead of bookkeeping code as required in trace scheduling. Another advantage of treegion scheduling is that it limits the enumeration of different control paths since treegion formation stops at merge points. In this chapter, we modify

the original TTS algorithm to be profile-independent since the objective here is to optimize the worst-case scenarios instead of optimizing the most frequently executed paths.

Designed as an EPIC approach, treegion scheduling exploits many architectural features of EPIC to improve ILP so that the execution time for multiple control paths (including those generating the WCET) is reduced simultaneously. The two most commonly used EPIC features are control speculation and predication. Both features suit the needs of real-time applications. Control speculation in general purpose computing would require recovery code generation for those instructions that could potentially cause an exception. Such a problem is simplified in real-time applications as we expect that real-time programs are well behaved and would not throw an exception in normal execution. As a result, there is no need to produce recovery code for control speculation because the deferred reporting of an exception [49],[65] is enough to report such a case when it really occurs. Predication, on the other hand, removes the execution time variability due to dynamic branch prediction. Treegion scheduling uses both features extensively and provides a unified framework for both of them in the scheduling process. In addition to speculation and predication, there are other EPIC features that facilitate real-time applications including static branch prediction hints, cache level specifications in loads and prefetches (i.e., the compiler scheme to control when the data to be cached in the memory hierarchy), and unbundling of branches as they reduce the variability of the dynamic execution time.

In summary, the high performance, low complexity, and compiler-controlled POE make EPIC architectures a good platform for real-time applications while the downside

of EPIC architectures, namely binary compatibility, is expected to be a less problem for real-time systems. Next, we will discuss how to use treegion-based scheduling and ILP optimizations efficiently to reduce the WCET of each task so as to improve the overall task-level schedulability.

4.3 Code Size Efficiency Based on Profile Independent Performance Bounds

As we discussed in Chapter 3, code size related ILP optimizations are shown to be very effective in improving ILP at the cost of static code size increase. Such ILP improvement can be used to reduce the WCET of a subtask/task/program. The reduction in WCET is important since (1) it can reduce the WCET of a task to make it meet its deadline specification; (2) it enables the processor to serve more tasks, thereby achieving better utilization; (3) the system can run at a lower frequency to save energy when enough slack can be produced. The static code size, as pointed out in Chapter 3, is also important as oversized programs can increase the system cost and lead to potential performance problems. Our goal is to achieve a good tradeoff between the WCET reduction and the static code size increase.

First, we extend the notion of code size efficiency defined in Chapter 3 to show the WCET reduction impact at the cost of code size increase. As discussed in Section 2.4, the actual WCET calculation requires time-consuming instruction scheduling and the LBWT reduction is used instead to evaluate the effectiveness of code optimizations. So, we define the *instantaneous code size efficiency* for each individual optimization instance as Equation 4-1.

$$Efficiency_{inst} = \frac{LBWT_{before\_individual\_application} - LBWT_{after\_individual\_application}}{code\_size_{after\_individual\_application} - code\_size_{before\_individual\_application}}$$

Equation 4-1

The numerator in Equation 4-1 represents the LBWT reduction resulting from the optimization and the denominator is the static code size increase. Note that the LBWT in Equation 4-1 can be defined at different levels to show the varying impact of a code optimization. For example, LBWT defined at a loop body level (i.e., a subtask) shows the optimization impact on reducing the WCET of the loop/subtask, and the LBWT at the task level reveals the task level impact of the same optimization.

4.4 Regulating the Code Size Related ILP Optimizations for Real Time Applications

We develop an algorithm as shown in Figure 4.1 to systematically regulate code size related ILP optimizations. Similar to the algorithm in Figure 3.4, the code size related ILP optimizations are performed in three steps. In the preparation step, the basic scheduling regions are formed (natural treegions in our case) and the optimization candidates are identified.

The optimization candidates are then treated differently based on their code size efficiency characteristics in Step 1 and Step 2. Optimizations with positive speedup and negative code size increase are examined first in Step 1 of the algorithm. Then, an iterative approach is used to selectively perform code-expanding optimizations, as shown in Step 2 of the algorithm. First, step 2a computes the efficiency of all potential optimization instances. Then, the best candidate is selected based on these efficiencies in

step 2b. Next, if the one with the best efficiency passes the feasibility check, it will be performed in step 2c. As one particular optimization may change the efficiency of another optimization or enable another optimization (e.g., one instance of tail duplication may enable a hammock to be constructed for if-conversion), a local efficiency update is performed in Step 2c if one optimization instance is performed.

Algorithm for regulating code size related optimizations in real-time applications

0. Form basic scheduling regions to facilitate LBWT computation and to identify program structures that are candidates for optimizations.
1. Perform code size reducing optimizations: ***if-conversion (or predication)***
    a. For a diamond / hammock structure, compute performance gains of if-conversion.
    b. If the if-conversion produces positive (or zero) LBWT reduction, perform it
    c. If the performed if-conversion results in a new diamond / hammock for its parent branch, continue to check this parent branch for if-conversion.
    d. Repeat step 1a – 1c, until no more diamond / hammock structures need to be checked.
2. Perform code size increasing optimizations: ***loop unrolling*** and ***tail duplication***
    a. Compute instantaneous code size efficiency for each loop unrolling / tail duplication candidate using Equation 4-1.
    b. Search the candidate list to find the one with the highest efficiency.
    c. If the selected candidate passes the feasibility check, perform the optimization and update the efficiency of candidates affected by the optimization. (The feasibility check may include code size constraints, register pressure, etc.)
    d. Repeat step 2a – 2c, until the overall code size reaches a limit or there are no more candidates.

Figure 4.1. The algorithm for regulating code size related optimizations for real-time applications.

Next, we use if-conversion as an example to see how we handle the branch misprediction penalty since it is not included in the original definition of LBWT in Section 2.3. For a conditional branch as show in Figure 4.2, the static branch prediction is determined as follows to minimize the WCET [3].

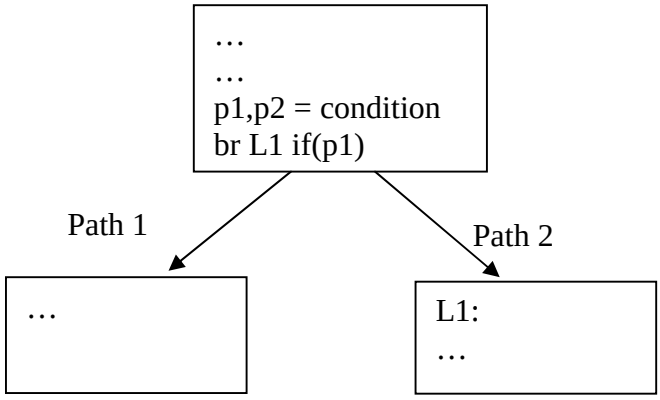


Figure 4.2. Predicting a conditional branch statically to minimize WCET.

If ($LBET_{path1} + LBWT_{base\_path1} > LBET_{path2} + LBWT_{base\_path2}$),

then the branch is predicted '*taken*'.

Otherwise,

the branch is predicted '*not taken*'.

Such static prediction favors the longer control path so that the misprediction penalty is imposed on the shorter path. Considering the overall WCET, such penalties can be hidden if the sum of the shorter path execution time and the misprediction penalty does not exceed the longer path execution time. If the shorter path and the longer path are more or less balanced (i.e., $LBET_{path1} + LBWT_{base\_path1}$ is close to $LBET_{path2} + LBWT_{base\_path2}$), the branch misprediction penalty will be imposed on the overall WCET. For such cases, if-conversion provides an effective way to remove branch prediction

related penalties if a diamond/hammock structure can be formed with the conditional branch. Figure 4.3 shows such an example, which is the same code example as in Figure 2.7.

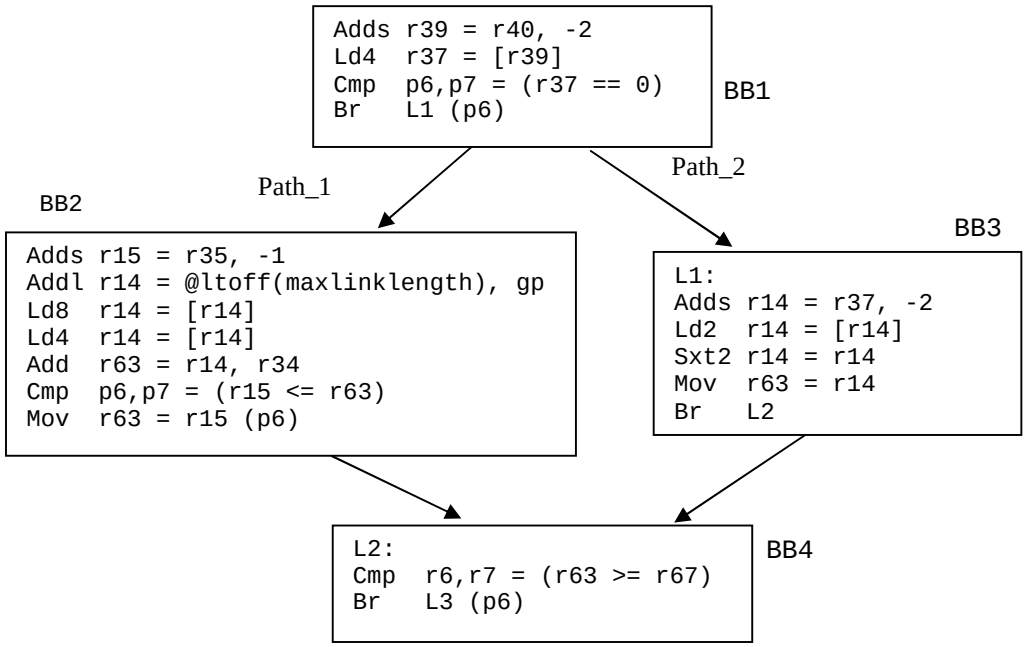


Figure 4.3. A diamond structure.

For a target 6-issue machine as used in Chapter 2, the LBWT for such a diamond structure is 9 cycles as computed in Section 2.4. The LBWT for path_1 is 8 cycles and the LBWT for path_2 is 5 cycles. When considering the static branch prediction (set as '*not taken*' as path_1 has longer execution time) and a 10-cycle misprediction penalty, the LBWT of this hammock is 10 + 5 + 1 = 16 cycles (the misprediction penalty is imposed on the originally shorter path).

With if-conversion, the instructions in BB2 and BB3 are predicated and the conditional branch is removed. A new basic/hyper block is formed containing BB1, BB2, BB3, and BB4. The LBWT now is 8 + 1 = 9 cycles using Equation 2-4 since this if-conversion does not result in any resource conflicts. So, such an if-conversion achieves a

7-cycle LBWT reduction and reduces the code size by 2 instructions (i.e., the conditional branch in BB1 and the unconditional branch in BB3).

Note that not all conditional branches can be if-converted. To remove the associated branch misprediction penalty, the conditional branch needs to form a diamond/hammock while its control paths have no other branches. For loop back branches, we can either choose ‘taken’ as the static prediction or we can assume ideal prediction by utilizing the loop count feature in the branch handling mechanisms of EPIC architectures [32].

4.5 Experimental Methodology and Results

As in Chapter 3, we implement our proposed algorithm in the LEGO compiler and the selected benchmarks from both the SPEC 2000 INT benchmark suite and the MiBench [27] suite are used as our workloads. As our objective is to reduce the WCET, we excluded the benchmarks containing recursive function calls or un-structural loops since they present obstacles for our current LBWT/WCET analysis. The selected benchmarks are first compiled into IA-64 assembly using the *gcc* compiler (version 3.1). As our focus is ILP optimizations, we use the level one optimizations provided by *gcc* to perform classical optimizations. For the workloads from MiBench, ‘small’ data input sets are used to determine the loop counts. For the benchmark *bzip2*, we use the reference input data set and skip the first 500 million instructions and profile the next 500 million instructions. In the experiments, after the ILP optimization phase, the code is scheduled using treegion scheduling and the WCET is computed.

Step 1 of the algorithm shown in Figure 4.1 regulates how code size decreasing optimizations, if-conversion in this chapter, are performed. Due to its code size reduction effects, any if-conversion, which produces positive (or zero) LBET reduction (i.e., positive speedups), will always be performed. As described in Section 4.4, we use static branch prediction to estimate the branch misprediction impact on LBWT/WCET assuming that each misprediction incurs a 10-cycle penalty.

As stated previously, *gcc* optimization level one is used to generate the IA64 assembly. However, the level one optimization of *gcc* also produces predicated instructions (i.e., the level one optimization performs if-conversion). So, in the first experiment, we modified *gcc* 3.1's source code to turn-off its if-conversion transformation and use the algorithm in Figure 4.1 (step 1) to perform if-conversion. The results in WCET reduction are shown in Figure 4.4. From Figure 4.4, it can be seen that aggressive if-conversion can reduce WCET significantly, up to 80% for the benchmark *adpcm* since its source code contains many 'if-then' and 'if-then-else' structures. In the benchmarks *rijndael* and *sha* (both are security benchmarks), the encryption/decryption kernel contains mostly straight-line instructions. Therefore, if-conversion yields negligible impacts on WCET reduction.

In the next experiment, we turn on gcc's if-conversion option to generate the assembly and then use our algorithm to perform if-conversion. The results shown in Table 4.1 indicate that our algorithm can improve upon *gcc*'s if-conversion algorithm.

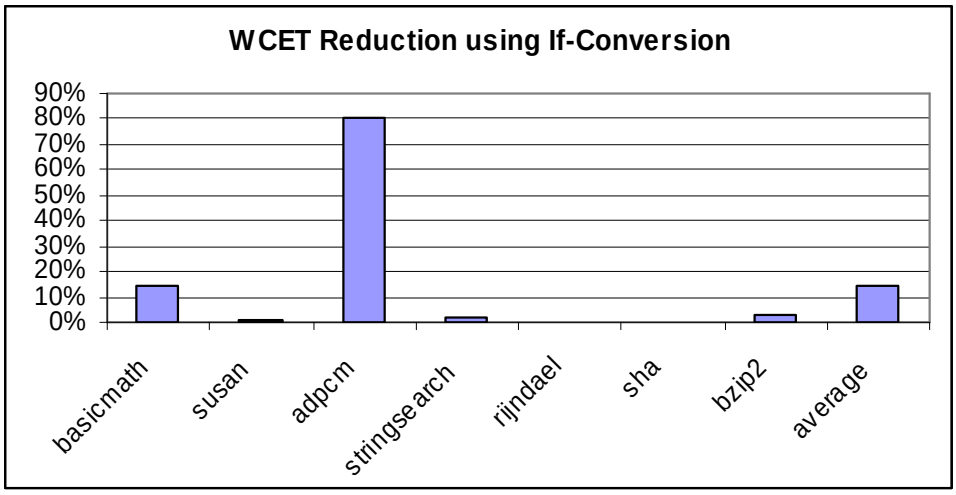


Figure 4.4. The WCET reduction using if-conversion.

Interesting observations can be made from Table 4.1. The first row in Table 4.1 reveals that *gcc* has removed a significant amount of conditional branches through predication, although the second row, which shows the number of existing conditional branches in each benchmark after *gcc*'s if-conversion, suggests that there still exist potential if-conversion candidates. Our algorithm examines those conditional branches and confirms that the majority of these conditional branches are not appropriate for if-conversion either due to complex CFGs, which inhibit diamond/hammock detection, or the detected diamond/hammock contains function calls, returns, or indirect branches (we excluded the case that both paths contain the same function calls or returns). In such cases, if-conversion may hurt WCET since if-conversion introduces more conditional function calls and returns, which in turn incur more branch misprediction penalties. For those if-convertible branches, our algorithm computes the LBWT reduction. Using the benchmark *adpcm* as an example, *gcc* converts 21 conditional branches and there remain 8 conditional branches in the program. Our algorithm finds that 4 of them do not form a diamond/hammock structure. For those that do form a diamond/hammock, 2 of them

have either a function call or a return instruction along one or both paths. For the remaining 2 conditional branches, both of them produce positive LBWT reductions.

Table 4.1. The if-conversion results

	basicmath	susan	adpcm	stringsearch	rijndael	sha	bzip2
If-conversions (by gcc)	3	72	21	11	13	3	113
Number of conditional br	18	325	8	50	56	14	487
If-conversions with pos. gain	0	30	2	0	0	0	23
If-conversions with zero gain	0	18	0	0	1	0	2
If-conversions with neg. gain	0	0	0	0	0	0	0
No if-conversion: complex CFG	16	123	4	43	37	10	358
No if-conversion: ret_call	2	154	2	7	18	4	104
Reduction in WCET (%)	0	0.07	25.65	0	0	0	1.40
Reduction in static code size (%)	0	1.6	1.85	0	0.06	0	0.66
Original static code size	643	5967	216	658	3664	409	7543

Next, we analyze the performance impact of if-conversion. In this experiment, we perform only the if-conversions that produce positive LBWT reductions. Although the number of these if-conversions seems limited (2 to 30, the third row in Table 4.1), additional WCET reduction (up to 26%) can be achieved, as shown in the 9th row of Table 4.1.

Finally, we analyze the code size reduction impact of if-conversion. We choose to perform if-conversions with positive or zero LBWT reduction in this experiment. Assuming each conversion saves two instructions in the IA-64 assembly, the overall code size reduction is computed and is shown in the 10th row of Table 4.1. Remember that this reduction is achieved on codes that have already been predicated by *gcc*. This shows that

our algorithm reduces code size by performing if-conversion more aggressively. From Table 4.1, it can be seen that if-conversion reduces static code size up to 1.85% (the benchmark *adpcm*) and 0.60% on average. Although these numbers seem to be trivial, in the next subsection, we will show that utilizing such a small amount of code size can lead to more WCET reduction.

Step 2 of the algorithm shown in Figure 4.1 regulates code size increasing optimizations (tail duplication and loop unrolling). It iteratively selects and performs the one instance of tail duplication or loop unrolling with the highest instantaneous code size efficiency. In this experiment, we examine the effectiveness of such an iterative approach. For each benchmark, we set the limit of code size increase as 5%, 10%, and 20% of the original size. (i.e., the optimization stops when the overall code size increase reaches the limit). The corresponding WCET reductions are shown in Figure 4.5.

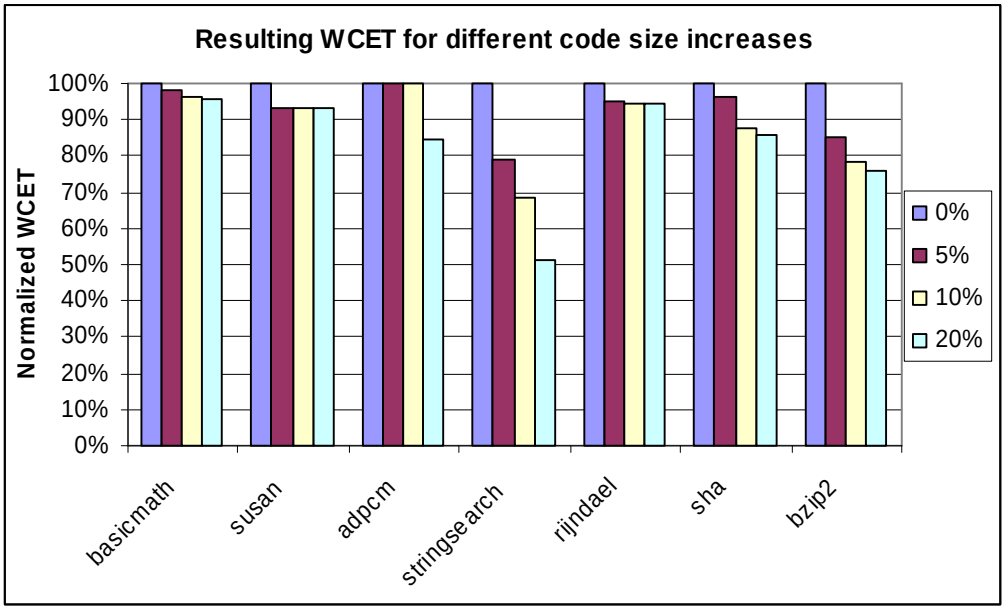


Figure 4.5. Resulting WCET for different code size increases.

From Figure 4.5, it can be seen that significant reductions in WCET are achieved from code size enlarging optimizations. The benchmark ***stringsearch*** shows the largest WCET reduction. Looking into its optimization process, we found that the gain is mainly from unrolling two frequently executed loops. These loops have no loop-carried dependence, thus producing large performance improvement from unrolling until the resource bound becomes the bottleneck. The benchmark ***basicmath***, on the other hand, shows the smallest WCET reduction. This benchmark contains several basic math functions, including ***SolveCubic***, ***usqrt***, and ***rad2deg***. The most frequently called function ***SolveCubic*** contains only one treegion; therefore it cannot be optimized further with either tail duplication or loop unrolling. As a result, this benchmark does not show big WCET reduction though other functions are highly optimized with unrolling and tail duplication. Also, from Figure 4.5, it can be noticed that for benchmark ***adpcm***, there is no reduction when the code size increase limit is set to 5% or 10% while it shows 18% reduction when the limit is 20%. The reason is that in this benchmark, the main loop body of the encoding contains a hammock. Duplicating the merge block of this hammock provides large performance gains but the size of this block is about 11% of its original size.

The ***diminishing returns*** phenomenon can also be observed for most benchmarks from Figure 4.5. Using the benchmark bzip2 as an example, the first 5% code size increase leads to a 15% WCET reduction while the next 5% (total 10%) code size increase leads to an additional 7% reduction and another 10% (total 20%) code size increase results in another 2% WCET reduction. This shows that during the iteration process of performing loop unrolling and tail duplication, the efficiencies of the initial

selections are much higher than the remaining ones. Other benchmarks show a similar trend and most of the WCET reduction is achieved in the first 5%-10% of code size increases. For the benchmark *stringsearch*, there exists significant WCET reduction until the code size increase reaches 50%, as seen in Figure 4.6.

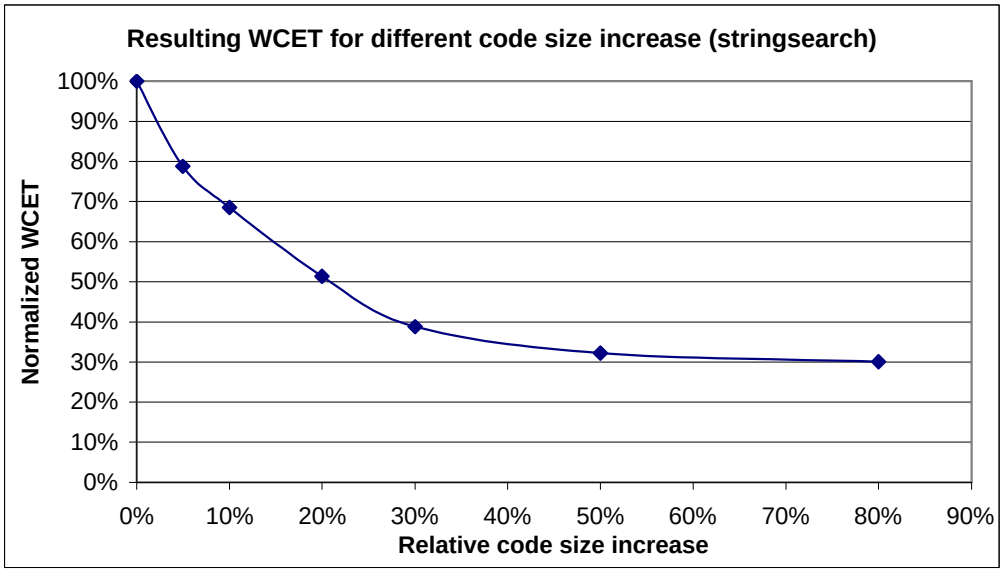


Figure 4.6. The diminishing returns exhibited from the benchmark *stringsearch*.

There are two fundamental reasons for the diminishing returns phenomenon as observed for general purpose computing in Section 3.5.3. First, based on the definition of code size efficiency, the effects of optimizations (LBWT/WCET reduction) occurring inside loop bodies are amplified by the factor of the loop count. The well-known '*90/10 rule*' points out that a small part of the static code (hot portions) consumes most of the execution time in general purpose computing. A similar rule also holds for real-time applications as most of the WCET is spent on a few heavily executed loops and functions called from these loops. After performing optimizations in these 'hot' portions of code, the remaining optimizations should have much lower efficiencies. Secondly, high efficiency also requires that the resulting code must have significantly reduced LBWT,

i.e., the optimization instance must reduce the DDG height without causing any resource conflict problems. This requirement filters the optimizations occurring in hot portions of a program.

One practical implication of this diminishing return phenomenon is that we can monitor the code size efficiency during the iteration process. If the highest candidate is below a threshold, we can expect that further optimization will have a minor impact on WCET reduction while involving large static code size increase.

4.6 Summary

In this chapter, we advocate using compile-time ILP optimizations and instruction scheduling to reduce the WCET. We propose the use of profile-independent performance bounds to evaluate the performance potential of ILP optimizations so as to avoid computationally expensive instruction scheduling. Then, we develop a general framework to selectively perform ILP optimizations based on their performance potential and the cost in code size increase. The experimental results show that by combining aggressive instruction scheduling and carefully performed ILP optimizations the WCET can be significantly reduced at a cost of minor static code size increase.

Combined with the discussion in Chapter 3, it can be seen that performance bound guided code size efficiency forms a systematic method for selectively performing code optimizations, and the effectiveness of such a method is seen for both general purpose computing and embedded computing, real-time computing in particular.

Chapter 5 Performance Modeling of Memory Latency Hiding Techniques

In this chapter, we discuss using performance bounds to model two related memory latency hiding techniques, address prediction based memory prefetching and value prediction, in memory-intensive workloads featuring heavy pointer chasing.

5.1 Introduction

The trends in contemporary microprocessor design, including fast clock speeds, deep pipelines [66], large window sizes [34],[39], aggressive out-of-order instruction execution, and wide fetch bandwidths [61], result in a tremendous ability to perform arithmetic computations (i.e., computation not involving slow memory operations such as cache misses). Therefore, for memory intensive workloads, especially those with heavy pointer chasing, it is more important to parallelize multiple cache misses than to overlap cache misses with other computations. For example, assuming the pointer-chasing code shown in Figure 5.1 results in many cache-misses due to traversing the linked list, these

cache-misses form a memory dependence chain due to the dependencies between the missing loads.

```
while (a != NULL) {
    //Processing the fields of a
    a = a->next;
}
```

Figure 5.1. A pointer-chasing code example.

As processing the linked-list takes little time compared to traversing the linked-list, the overall execution time is mainly determined by resolving such a memory dependence chain of missing loads. To reduce the time of serving these dependent cache misses, different techniques have been proposed. Memory prefetching [5],[14],[33], based on address prediction of the missing loads, tries to bring the data close to the processor (e.g., L1 or L2 D-Cache) long before the missing load executes so that the miss latency can be overlapped either with computation or with previous load misses. In the code example in Figure 5.1, every successful address prediction has the potential to eliminate one cache miss.

Value prediction [21],[43],[44], which relies on the predictability of the destination value of an instruction (e.g., the load value) rather than the load address, enables dependent computations to be executed speculatively while the missing load is being served. However, for pointer-chasing codes, the predictability of load values can be viewed as *equivalent* to the predictability of load addresses since one load address is simply the previous load's value plus a constant offset. While value prediction was proposed originally to break true data dependencies as an instruction-level-parallelism (ILP) optimization, we advocate that the true merit lies in its ability to *enhance the*

memory-level-parallelism (MLP) by overlapping multiple outstanding load misses. In the code example in Figure 5.1, assume that the instruction window contains five iterations of the loop and all five pointer-chasing loads will miss in the data cache. Also, we assume that one of these five missing loads' values is predictable, say the second missing load (i.e., the address of the third missing load is predictable). Predicting the value of the second missing load enables two of its dependent loads (the third and fourth missing loads in this example) to be overlapped with the first and second missing loads. As a result, a single value prediction can reduce the number of cache misses by 2 -- much better than what would be achieved using a prefetch with the same predictability.

The above simple pointer-chasing code illustrates that value prediction can be more effective in overlapping cache misses and increasing memory level parallelism (MLP). In this chapter, we introduce a formal analytical model using performance bounds to evaluate and compare the performance potential of both prefetching and value prediction. The target workload is *memory intensive applications with heavy pointer chasing*. This analytical model reveals the capability of each technique in hiding cache miss latencies through MLP utilization. Important observations are drawn from the model: while prefetching is generally effective for short memory dependence chains, value prediction has ***better*** potential for long dependence chains. For a long dependence chain due to pointer chasing, the performance difference between value prediction and prefetching scales proportionally with the prediction accuracy, the memory dependence chain length, and load miss penalties. Since the chain length scales with the effective instruction window size and miss penalties scale with fast processor clock speed, the

model shows that value prediction is a very powerful technique for improving MLP in high performance microprocessors.

The rest of this chapter is organized as follows. Section 5.2 discusses the performance modeling of memory prefetching. Section 5.3 contains the modeling of value prediction. The performance comparison of the two is in Section 5.4. Finally, Section 5.5 summarizes and motivates novel techniques to improve MLP more effectively.

5.2 Performance Modeling of Memory Prefetching

For workloads with heavy pointer chasing, the memory dependence chain of missing loads dominates the overall execution time since other computations are either overlapped with the memory access latency or only accounts for a small portion of the overall execution time. Instead of relying solely on simulation, we use performance bounds to evaluate the performance potential of memory latency hiding techniques. For a memory dependence chain containing N dependent, missing loads (which we call *a dependence chain of length N*), a lower bound of execution time (LBET) is defined as the time to resolve all these missing loads:

$$\text{LBET}_{\text{original}} = N * Miss\_latency. \qquad \textbf{Equation 5-1}$$

In this model, we use the same miss latency to model the penalty of all missing loads. For a memory hierarchy with multiple cache levels, the miss latency varies at each level. As the miss latency at a higher cache level (e.g., L0 or L1) can usually be hidden successfully with out-of-order execution or aggressive instruction scheduling [69], we

choose to use this memory dependence chain to model a sequence of cache misses at a lower level cache (e.g, a sequence of dependent L2 misses).

To model the performance potential of memory prefetching, we assume that if the address of a missing load is predictable (i.e., the missing load is *prefetchable*), then a prefetch can be triggered early enough so that the miss latency is hidden completely. Such assumptions favor the results of prefetching, but do not affect our conclusions. Based on this idealistic assumption, if *K* missing loads along the chain can be prefetched, the performance bound is then the time to resolve the remaining (*N*-*K*) load misses:

$$LBET_{prefetch\_K} = (N - K) * Miss\_latency. \qquad \textbf{Equation 5-2}$$

In other words, prefetching *K* loads collapses a chain of length *N* into a chain of length (*N*-*K*). For example, consider the pointer chasing code "*a*->*b*->*c*->*d*->*e*", where *a*-*e* are loads, which results in four dependent missing loads. Prefetching any of them will reduce the length of the chain to 3, as shown in Figure 5.2.

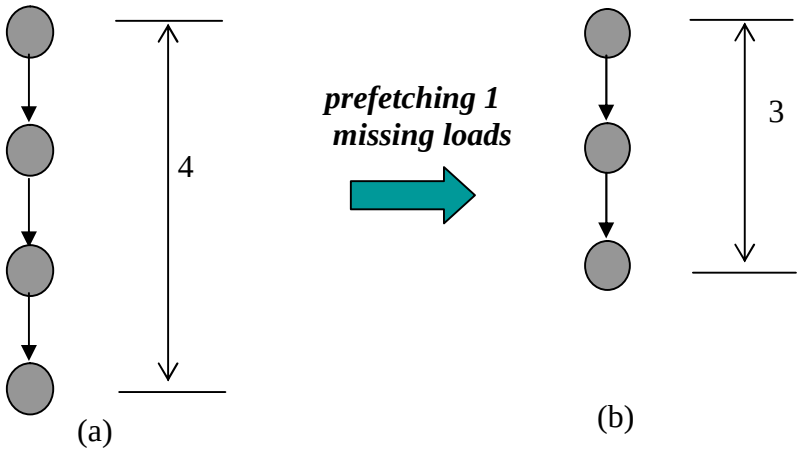


Figure 5.2. (a) The code 'a->b->c->d->e' resulting in a memory dependence chain of 4 missing loads; (b) Prefetching 1 missing load along the chain reduces the chain length by 1.

The model can also be extended to include the impact of load address mispredictions. If the prefetch address prediction accuracy is *x*% (assuming the same accuracy for all predictions for simplicity), the performance bound is the weighted sum of successful prefetching and prefetching with mispredicted addresses. Assuming prefetching a mispredicted address has little impact on the overall performance, the performance bound can be computed as:

$$\text{LBET}_{\text{prefetch\_}K\_accu} = \text{LBET}_{\text{prefetch\_}K} * x\% + \text{LBET}_{\text{original}} * (1 - x\%) \quad \textbf{Equation 5-3}$$

For a special case $K = 1$, i.e., prefetching one missing load, the performance bound is $N * \textit{Miss Penalty} - \textit{Miss Penalty} * x\%$.

5.3 Performance Modeling of Value Prediction

Predicting the value of a single missing load along a memory dependence chain, say the i^{th} load, breaks the dependence chain into two shorter ones. The performance bound is then determined by the longer one of the resulting two shorter chains, one with the length i and the other with the length $(N-i)$. Thus, the performance bound can be computed as:

$$\text{LBET}_{\text{prediction\_1}}(i) = \max\{i * \textit{Miss\_Penalty}, (N-i) * \textit{Miss\_Penalty}\} \quad \textbf{Equation 5-4}$$

As can be seen from Equation 5-4, unlike prefetching, the performance bound based on value prediction is dependent on where the prediction is made along the chain. Using the example in Figure 5.2, we can enumerate the predictions made for all different missing loads along the dependence chain, as shown in Figure 5.3. Figure 5.3a shows the memory dependence chain. In Figure 5.3b, predicting the first missing load enables the second load to be executed speculatively, therefore overlapping the memory access

latency of these two loads. Predicting the value of the second load breaks the chain more evenly and results in more overlapping of missing loads (i.e., more memory level parallelism), as shown in Figure 5.3c. Figure 5.3d shows that predicting the third load only enables the fourth load to be overlapped. Predicting the value of the fourth load does not reduce the chain length assuming that there are no missing loads dependent on it.

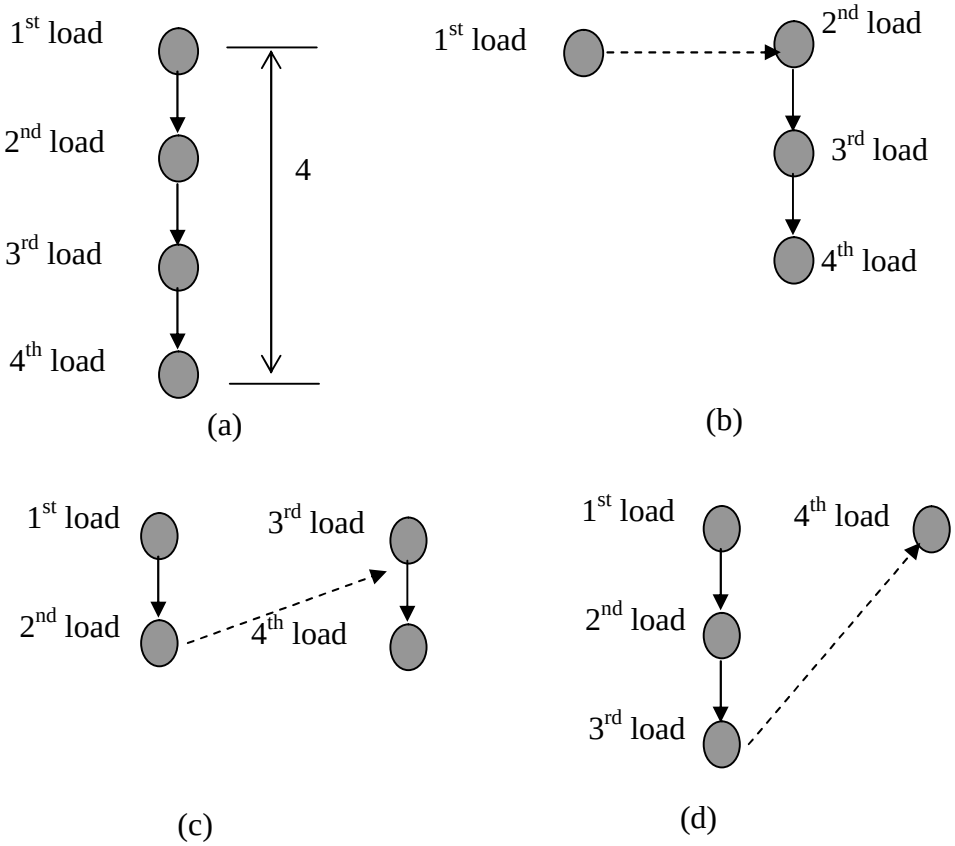


Figure 5.3. (a) A memory dependence chain of 4 miss loads; (b) Predicting the value of the first missing load; (c) Predicting the value of the second missing load; (d) Predicting the value of the third missing load.

As the performance bound of predicting one load along the memory dependence chain depends on which load is predicted, we can use a *probabilistic* approach to model

this effect. Assuming $p(i)$ is the probability of a prediction happening at the i^{th} missing load along the chain, the performance bound can be derived as follows.

$$\text{LBET}_{\text{prediction\_1}} = \sum_{i=1}^{N} p(i) * \text{LBET}_{\text{prediction\_1}}(i) \qquad \textbf{Equation 5-5}$$

For a uniform distribution of $p(i)$, (i.e., $p(i) = 1/N$), Equation 5-5 can be simplified into Equation 5-6 when N is odd:

$$\text{LBET}_{\text{prediction\_1}} = 2 * \sum_{i=1}^{\frac{N-1}{2}} [p(i) * (N\text{-}i) * \textit{Miss\_Penalty}] + p(i = N) * N * \textit{Miss\_Penalty}$$

$$= (\frac{3}{4} * N + \frac{1}{4N}) * \textit{Miss\_Penalty} \qquad \textbf{Equation 5-6}$$

Similarly when N is even, $\text{LBET}_{\text{prediction\_1}} = (\,3/4 * N\,) * \textit{Miss\_Penalty}$.

Based on this derivation, one very interesting observation is that predicting a single load has a similar effect to reducing the chain length from N to ¾*N. Thus, predicting K loads along the chain would reduce the chain to $(¾)^K * N$, i.e., $\text{LBET}_{\text{prediction\_k}} \approx (¾)^K * N * \textit{Miss\_latency}$ as one prediction usually would not affect the predictability of other instructions. When taking the prediction accuracy (which is the same accuracy for all predictions for simplicity purposes), x%, into account, the performance bound becomes Equation 5-7, assuming the misprediction penalty is small compared to the load miss latencies.

$$\text{LBET}_{\text{prediction\_k\_accu}} = \text{LBET}_{\text{prediction\_k}} * x\% + \text{LBET}_{\text{original}} * (1 - x\%) \qquad \textbf{Equation 5-7}$$

5.4 Comparison between Prefetching and Value Prediction in Hiding Miss Latencies

In this section, we compare the performance potential of prefetching and value prediction. Since predicting multiple values or addresses along the chain are equivalent to making single predictions multiple times sequentially (since making one prediction in the memory dependence chain usually does not affect the next prediction), we focus on the case of predicting a single value or address along the chain.

As discussed in the previous sections, prefetching a single missing load (i.e., predicting the address of one missing load) reduces the chain length by 1 (from N to N-1) while predicting the value of a single load has the potential to reduce the chain length by ¼*N. As discussed in Section 5.1, if the memory dependence chain is due to pointer chasing, then the predictability of the i^{th} load value is equivalent to the predictability of the address of the $(i+1)^{th}$ load. Thus, the performance difference between single prefetching and single value prediction is (1 – ¼*N) * *Miss_latency* * x%. As a result, we can see that if N < 4, then ¼*N < 1, which implies that prefetching outperforms value prediction for ***short*** memory dependence chains. When $N \geq 4$, then ¼*N ≥ 1, which shows that value prediction has better performance potential for memory dependence chains containing more than 4 dependent missing loads. Moreover, the performance difference is proportional to chain length, cache miss latency, and prediction accuracy, i.e., value prediction is more superior to prefetching for higher miss latencies and better prediction accuracies. This conclusion is somewhat surprising as prefetching is a widely accepted technique to overcome the memory gap while value prediction, proposed as an

instruction-level-parallelism (ILP) optimization, has not found its application in current processor design since ILP is not yet limited by true dependencies. Here, we argue that the most significant merit of the value prediction technique lies in its ability to *enhance the memory-level-parallelism (MLP) instead of improving the ILP for memory intensive workloads*.

Although we establish that value prediction has greater performance potential based on the analytical model, we need to examine it more carefully to understand why it produces such potential. In order to do that, we take another look at the example in Figure 5.2 and Figure 5.3 and replicate the dependence chain in Figure 5.4a. Assume the third missing load along the dependence chain is prefetchable, which also means that either the address of the third load is predictable or the value of the second load is predictable. The prefetching scheme uses this predicted address to execute a data fetch from memory and this fetch latency can be overlapped with outstanding misses of the first load, as shown in Figure 5.4b. The value prediction scheme uses the prediction of the second load (or the address of the third load) similarly to bring in the data. However, it takes this one step further by utilizing the fetched data to execute another dependent load (the fourth load in this example) so that the fourth load can be overlapped with the second load, as shown in Figure 5.4c.

From this example, we can see that the *key* reason that value prediction provides more MLP than prefetching is that it *uses* the fetched data to enable more dependent missing loads to be executed. If the prediction is correct in the first place, such speculative execution *propagates the predictability* even though the dependent loads are not predictable (but it is computable based on the previous predictions). In this example,

even if the value of the third load is not predictable, the data of the fourth load can still be fetched. This observation also explains why a recently proposed technique, called stateless, content-directed data prefetching [17] works better than traditional prefetching schemes. Content-directed data prefetching analyzes the content of the fetched data block to check whether the data could potentially be a pointer de-reference address. If so, it will attempt to fetch the data from this address as well. Value prediction, compared to this content-directed approach, uses the fetched data more judiciously by following the code semantics so that it uses the resources more efficiently and has fewer chances to pollute the cache.

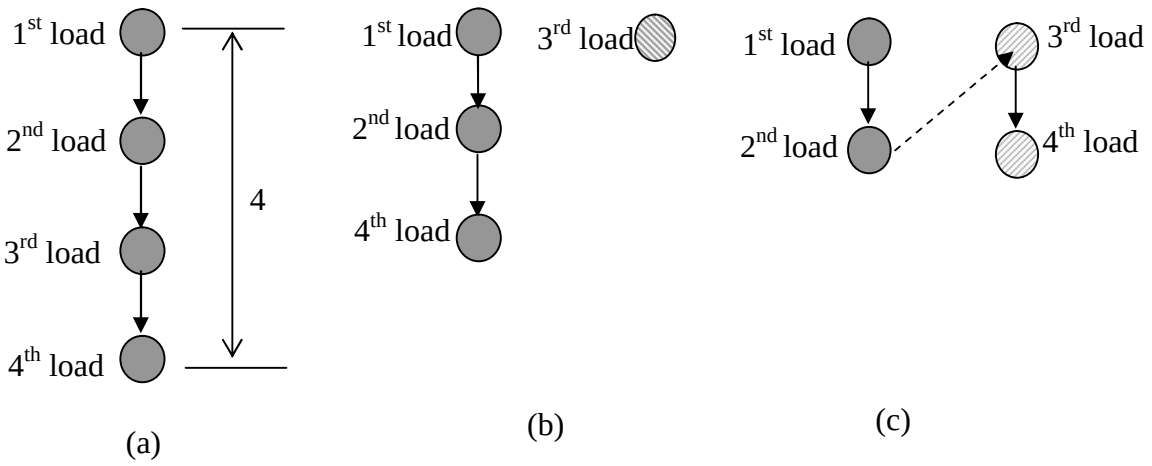


Figure 5.4. A memory dependence chain of 4 miss loads; (b) prefetching the third load; (c) value predicting the second load.

5.5 Summary

An analytical model is developed in this chapter to model the performance potential of prefetching and value prediction for memory intensive workloads. It is established that for pointer-chasing codes with long dependence chains (e.g., chasing a

link-list), value prediction introduces better MLP utilization and outperforms the prefetching technique given the same predictability model. The key reason for such success is that the fetched data is not only placed in the cache but also used to drive the dependent loads. The performance model also shows that the performance difference between value prediction and address prediction based prefetching scales with the chain length, the memory access latency, and prediction accuracy, thus making it compatible with trends in current microprocessor design.

Based on these important observations, the following interesting directions are worth examining to hide memory access latencies more effectively.

- Designing more powerful value/address prediction techniques to break memory dependence chains more aggressively.
- Combining both prefetching and value prediction can potentially provide better results, since prefetching works well for short chains and value prediction is better for longer chains.
- Using profile information: the analytical model in this chapter is based on several assumptions: prediction happens along a chain in a uniform distribution and all predictions have the same accuracy. Profile-based analysis can refine the model so that the compiler can use the prediction and/or prefetch techniques more effectively.

Chapter 6 Enhancing Memory Level Parallelism via Recovery-Free Value Prediction

Chapter 5 establishes that value prediction has good potential for hiding memory access latencies for pointer chasing, memory intensive workloads. In this chapter, we propose a cost-effective approach based on value prediction to speculatively parallelize sequential cache misses, thereby increasing memory-level parallelism (MLP).

6.1 Introduction

As discussed in Chapter 5, the contemporary microprocessors have tremendous capabilities in performing arithmetic computations (i.e., the computation not involving slow memory operations such as cache misses). Therefore, for memory intensive workloads, it becomes more important to parallelize multiple cache misses than to overlap cache misses with arithmetic computations.

In this chapter, we propose a novel technique to parallelize sequential cache misses speculatively. The target workloads are memory intensive benchmarks with heavy pointer chasing. The idea is developed upon value prediction [21],[43],[44], which was originally proposed as an instruction level parallelism (ILP) optimization to break true data dependencies in computations. Since the data dependence between pointer chasing loads enforces the sequential execution, value prediction has the ability to parallelize these loads, thereby increasing the memory level parallelism (MLP), as highlighted in Chapter 5.

Since we focus on using value prediction to increase MLP, the hardware overhead to support value prediction and value speculative execution can be significantly reduced. In this chapter, we propose to use value prediction only for prefetching so that the complex value prediction validation and misprediction recovery mechanisms are avoided and only minor changes in the hardware are necessary. Unlike the traditional value prediction schemes, where speculative results are committed when the correct prediction is made, the speculative results in our scheme are only used for prefetching and will not be committed. In a different point of view, one can think of the speculative execution in our approach as a speculative pre-execution scheme, which requires neither explicit pre-execution thread generation nor multi-threading support. Another important aspect is that the same hardware changes in our scheme also enable aggressive memory disambiguation to break alias (i.e., the load-after-store) dependencies. Such disambiguation is used for prefetching and is also recovery free.

The experimental results, based on a set of SPEC2000 benchmarks [30] and Olden benchmarks [12] including both computation-intensive and memory-intensive

benchmarks, show significant speedups resulting from breaking both true dependencies and alias dependencies between memory operations. Such speedups also scale well with the current trend in microprocessor design.

The remainder of this chapter is organized as follows. Section 6.2 addresses related work. Section 6.3 illustrates the performance potential of breaking memory dependencies to enhance MLP. Section 6.4 presents the details of our proposed approach. The experimental methodology is contained in Section 6.5, and the results are given in Section 6.6. Section 6.7 discusses the limitations of our proposed scheme. Section 6.8 summarizes the chapter.

6.2 Related Work

Due to the speed gap between the processor core and the memory, hiding memory access latency has been an active research topic. One well established solution is memory prefetching, and the majority of work is based on address prediction [5],[33]. One recently proposed scheme [17], named *stateless, content-directed prefetch*, improves upon prior techniques by examining the prefetched data to check whether the data could potentially be a pointer de-reference address. If so, the content will be used as the address for the next prefetch. Compared to this scheme, our proposed technique uses the fetched data to compute pointer chasing load addresses based on code semantics, thereby having fewer chances to fetch the wrong data and pollute the cache.

Another way to hide memory latency is based on the concept of pre-execution/pre-computation. Both hardware-based and software-based schemes [16],[47], [62],[69],[79] have been proposed for this purpose. As will be discussed in Section 6.4,

our recovery-free value prediction scheme is similar to the pre-execution paradigm although our approach requires neither explicit thread generation nor multi-threading support. Also, as pointed out in [69], the pre-computation thread is more effective when used to prefetch the critical pointer chasing loads in the loop control than to prefetch loads in the loop body. A similar observation can be made for our proposed scheme since predicting pointer-chasing loads in the loop control can overlap the execution of multiple iterations and result in more latency hiding. Runahead execution [18],[57] is another form of pre-execution without multithreading support. During execution, if the processor is stalled due to a cache miss, the current execution state will be checkpointed and the processor will enter the runahead mode to pre-execute the *independent* instructions following the blocking instruction. The purpose of the pre-execution is to prefetch the (future) data into cache. When the cache miss is repaired, the processor goes back to normal mode and re-executes these pre-executed instructions. In an out-of-order processor, runahead execution can achieve similar performance to one with a much larger instruction window. Our proposed scheme and runahead execution can be mutually beneficial as our scheme tries to pre-execute the *dependent* operations of a blocking instruction. Also, as discussed in Section 6.3, the large instruction window achieved by runahead execution provides better chances for our scheme to improve MLP.

Value prediction was proposed originally as an ILP optimization technique [21],[43],[44],[64]. Using value prediction to hide load forward latencies is studied in [11]. By correctly predicting the value of a load instruction, dependent instructions can avoid stalling during the time that the load executes. Address prediction for prefetching is proposed in [24]. Based on address prediction, data is prefetched and saved in a special

buffer (called the Memory Prefetch Table) and used as the value prediction of the load. Our proposed approach is different from these previous works in that we use value prediction only for prefetching, thereby avoiding complex validation and recovery hardware and the associated recovery penalties. Also, our approach leverages aggressive memory disambiguation for prefetching. As pointed out in Section 6.3, it is very important to break *both* true and alias dependencies in order to increase MLP. Another important aspect is that our scheme distinguishes the value speculative execution from normal execution so that we only use the un-speculative results to update the value predictor, thereby being able to achieve better prediction results.

6.3 Breaking Memory Dependencies to Enhance MLP

Values produced by individual instructions exhibit locality [64] and different value prediction schemes are proposed to exploit such locality to break true data dependencies [21],[43],[44]. In a typical value prediction/speculation scheme proposed for a superscalar processor, the prediction of an instruction enables its dependent instructions to be executed speculatively. If the prediction turns out to be correct, these instructions will commit their speculative results so that the processor makes faster forward progress by hiding the latency of value speculative computation in the un-speculative computations. If the prediction is wrong, however, a recovery scheme is necessary to squash the speculative results and to re-execute these affected instructions with correct data. Such a recovery mechanism, especially selective reissuing, incurs expensive hardware overhead and recovery latency penalties [77].

As discussed in Chapter 5, for memory intensive workloads with heavy pointer chasing, sequential cache-misses resulting from pointer chasing code structures dominate the overall execution time. These cache-misses form a memory dependence chain since one missing load's address is dependent on the previous missing load's value. Taking a frequently executed code segment from the benchmark *mcf* as an example, shown in Figure 6.1, the profile information indicates that the pointer chasing codes '*node->child*', '*node->basic_arc->cost*', and '*node->pred->potential*' result in many cache misses. The memory dependence chains formed by these missing loads are shown in Figure 6.2.

```
while( node )
{
   if( node->orientation == UP )
      node->potential = node->basic_arc->cost + node->pred->potential;       // (Nodes
1,2,3,4)
   else /* == DOWN */
   {
      node->potential = node->pred->potential - node->basic_arc->cost;
      checksum++;
   }
   tmp = node;
   node = node->child;                                                    //  (Nodes 0, 5)
}
```

Figure 6.1. A code segment in the benchmark *mcf* (in function *refresh_potential*) resulting in many cache-misses.

In Figure 6.2a, the dependence chain is based on a single iteration of the *while* loop in Figure 6.1, where nodes 1 and 2 correspond to two dependent missing loads from '*node->basic_arc->cost*'. Nodes 3 and 4 correspond to '*node->pred->potential*'. Node 5 corresponds to '*node->child*' and node 0 is the same load '*node->child*' from the previous iteration. Figure 6.2b shows the dependence chain when the loop is unrolled multiple times. The solid arrow in Figure 6.2 represents the true data dependence and the

dashed arrow represents the alias dependence between missing loads. Although the alias dependence exists between a store and a following load instruction, we use the same term to model the dependence between two missing loads when one or more store instructions exist between them and one of these stores is dependent on the first missing load. Here, it needs to be pointed out that alias dependencies span multiple iterations, e.g., there exists an alias-dependence between node 2 in the first iteration and all the loads in later iterations, though not shown in Figure 6.2b for conciseness. Also, note that in the memory dependence chain, only missing loads are included as other instructions such as stores, adds, branches, and loads that hit in the cache are not long latency operations.

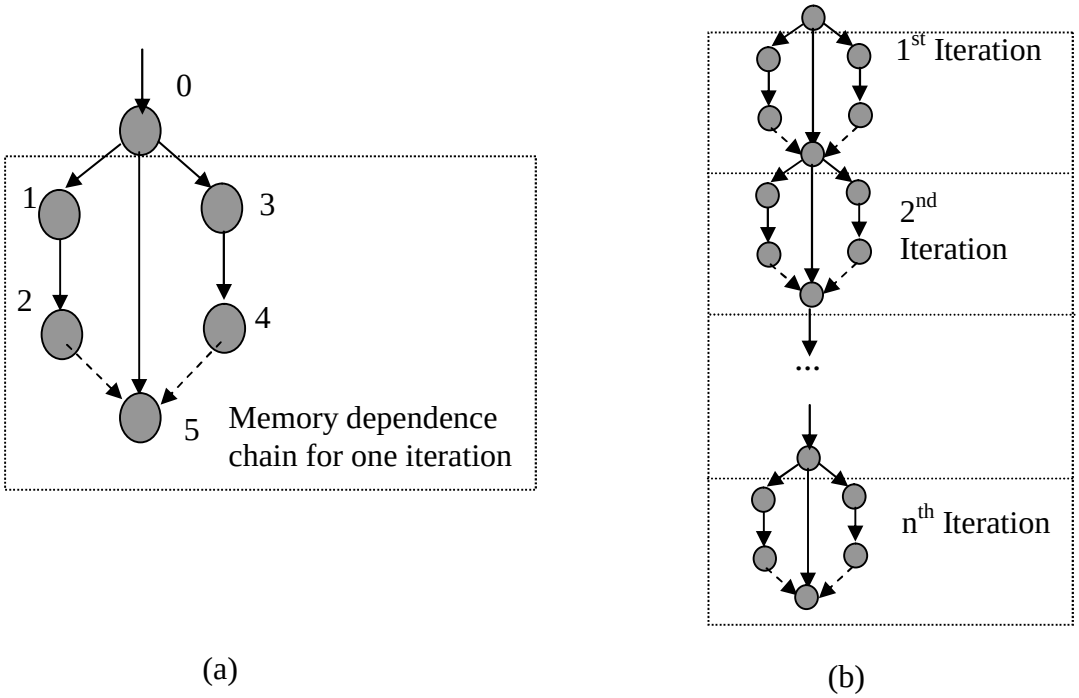


Figure 6.2. The memory dependence chain based on the code in Figure 6.1. (a) The dependence chain for a single iteration. (b) The dependence chain for multiple iterations (alias dependence among different iterations are not shown for conciseness).

From this example, we can see that both true data dependency and alias dependency enforce the sequential execution of the missing loads, resulting in long execution time. In order to process these cache misses in parallel (i.e., to increase MLP), both dependencies need to be broken. While aggressive memory disambiguation can minimize alias dependency, value prediction can be used to break true data dependency. In this example, memory disambiguation removes the dependence of node 5 on nodes 2 and 4 in Figure 6.2a, thus exposing the critical path of executing the loop as chasing the pointer '*node->child*' (i.e., node 5). If a correct prediction can be made for this load, the execution of multiple iterations of the loop can be overlapped, as shown in Figure 6.3, where predicting the value of the pointer chasing load (node 5' in Figure 6.3) in the second iteration enables the third and the fourth iterations to be executed speculatively so that their miss latencies are overlapped with the first and the second iterations. As a result, the long miss latencies in the third and the fourth iterations can be completely hidden if a correct value prediction is made.

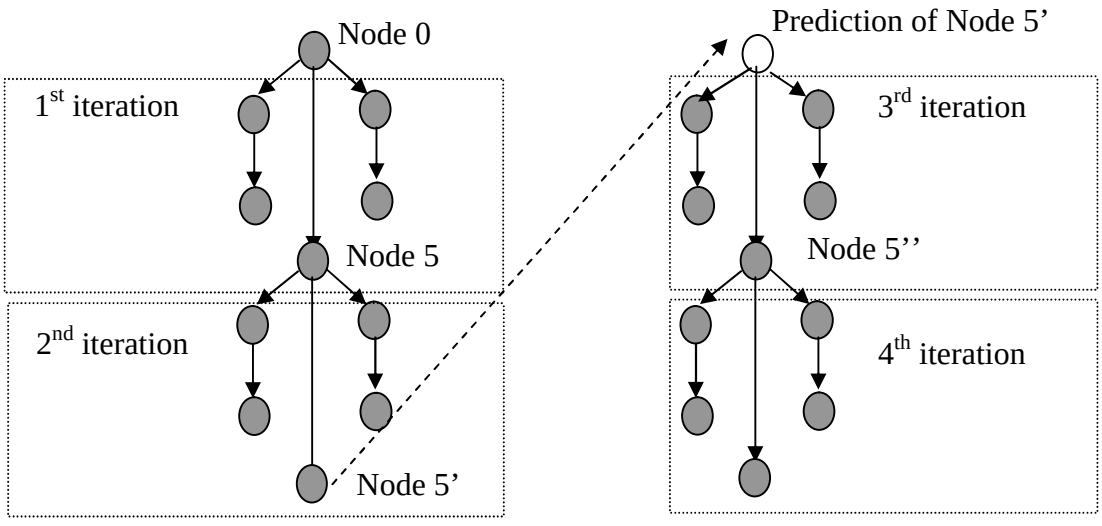


Figure 6.3. Predicting the value of Node 5' enables overlapping of cache misses in different iterations.

The example in Figure 6.3 illustrates that the effectiveness of value prediction in breaking the true memory dependence chain so that sequential cache misses can be processed in parallel and MLP can be enhanced. Such effectiveness is affected by several characteristics of the memory dependence chain. The first is the length of the memory dependence chain. In the example in Figure 6.3, the instruction window size determines how many iterations of the loop can be unrolled dynamically. If the instruction window can only hold two iterations of the loop, the speculative execution of the third and the fourth iterations is impossible when they are not fetched into the pipeline. The second is which missing load along the dependence chain is predicted. In the example in Figure 6.3, it can be seen that predicting the value of Node 5' can overlap more cache misses than predicting Node 5 or Node 5''. The third is the predictability of these missing loads' values since more accurate prediction will result in more useful speculative executions. These characteristics are examined in Chapter 5 using an analytical model of value prediction in enhancing MLP. It is found that value prediction can be more effective than traditional address prediction based prefetching techniques for the same predictability model. The main reason is that while prefetching techniques only bring the data close to the processor (e.g., the L1 D-cache), value prediction takes one step further by using the fetched data to drive other dependent load instructions to be executed early. In the example in Figure 6.3, it can be seen that predicting the value of Node 5' is equivalent to predicting the address of the dependent loads (e.g., Node 5'') since the only difference is a constant offset. So, using address prediction based prefetching, the miss latency of Node5'' can be hidden if the prefetch is triggered early enough. Value prediction, on the

other hand, not only fetches the data of Node 5'' but also uses the fetched data to execute other dependent instructions (i.e., the missing loads in the fourth iteration) even if their addresses/values are not predictable. As a result, value prediction is capable of hiding much more miss latencies. The analytical model also shows that the effectiveness of value prediction is proportional to the memory dependence chain length, the value prediction accuracy, and the cache miss latencies. Since the chain length scales with the effective instruction window size and miss penalties scale with fast processor clock speed, we argue that value prediction is a very powerful technique to improve MLP for future high performance microprocessors.

6.4 Recovery-Free Value Prediction

As discussed in Section 6.3, value prediction has great potential to enhance MLP by overlapping otherwise sequential cache misses. To implement such a technique, however, complex hardware support is necessary to validate the prediction and to perform recovery from value mispredictions. As discussed in Section 6.1, current microprocessors can execute computations very fast as long as slow memory operations (e.g., cache misses) are not involved. So, unlike previously proposed value prediction schemes [21],[43],[44], we propose to use value prediction only for prefetching so that there is no need to validate a prediction or to perform recovery from mispredictions. Using the example in Figure 6.3, based on the prediction of node 5', the third and the fourth iterations of the loop are executed speculatively. Unlike the traditional value prediction schemes, the speculative results will not be committed in our approach and the only purpose of such speculative execution is to bring the data to the L1 data cache. As a

result, even if the prediction is correct, the third and the fourth iterations of the loop will be executed again (un-speculatively) in our proposed scheme. We expect that such re-execution will be very fast since the cache accesses in these iterations will hit in the L1 data cache (as the data have already been fetched during speculative execution if the prediction is correct). So, compared to traditional value prediction schemes, our technique trades a small penalty of re-execution in the case of correct value prediction for much smaller hardware overhead. In the case of a value misprediction, both traditional schemes and our proposed scheme will result in polluting the data cache while our scheme incurs *no* recovery penalties. Another interesting point is that the same hardware changes required in our scheme also enable aggressive, recovery-free memory disambiguation for prefetching as a byproduct, therefore is capable of delivering higher performance improvement.

To support recovery-free value prediction, only minor hardware changes are necessary. We present our proposed design based on a MIPS R10000 style microarchitecture [73], which has a 7-stage pipeline as shown in Figure 6.4. There are four key changes to the hardware, presented as follows.

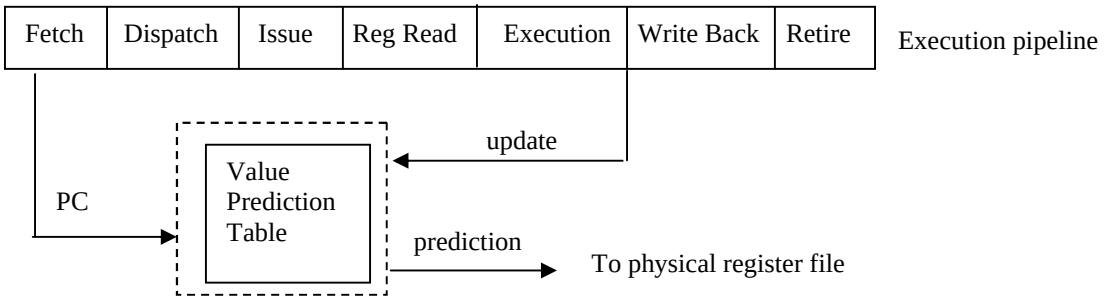


Figure 6.4. The execution pipeline.

First, a value predictor is included in the front-end of the processor and is indexed with the *pc*, as shown in Figure 6.4. The design of a high accuracy value predictor is out of the scope of this chapter and we use a simple stride value predictor [21],[43],[64] to show the effectiveness of our technique though a more powerful predictor [68],[74] can lead to a higher performance improvement.

Secondly, two flag bits are added to control value speculative execution. One flag bit, called value prediction speculative (*vp*), is added to every entry of the issue window or RUU. The other flag bit, called value prediction ready (*vp_ready*), is added for each register in the physical register file. When a confident value prediction is made at the dispatch stage, the *vp_ready* bit is set for the destination register and the predicted value is written to the physical register file. At the issue stage, if the source registers of an instruction are ready, it will be issued un-speculatively and the execution result will be used to update the value predictor. If source registers are not ready but the *vp_ready* bits for these source registers are set (i.e., the values of these physical registers are either predicted or computed using previous predictions), the instruction is issued speculatively provided that there are unused issue bandwidth and function units. When an instruction is issued speculatively, the corresponding *vp* flag in the issue queue is set to prevent the same instruction from being issued speculatively more than once since we do not need the same data to be prefetched more than once. Speculatively issued instructions will remain in the issue queue until they are issued un-speculatively later with (un-speculative) ready source registers. When a speculatively issued instruction finishes, it writes back the speculative results to the physical register file and sets the corresponding *vp_ready* bit to enable dependent instructions to be executed speculatively. Writing the

speculative results to the physical register file won't affect the correctness of program execution since the physical register will be overwritten by the un-speculative execution of the same instruction. In the case when the speculative result arrives later than the un-speculative result, it is simply dropped.

Thirdly, the instruction selection logic is modified so that it prioritizes the issue of un-speculative instructions and prohibits the speculative execution of store and branch instructions. In such a way, speculative execution will not compete with normal execution for resources and it only affects normal execution through the data cache.

Fourthly, to break the alias (i.e., load-after-store) dependencies, the *vp* flag is set for the load instructions that are stalled due to prior unresolved store addresses. Then, these load instructions can be issued speculatively as if they were based on predicted values. Therefore, no alias dependencies are enforced. This aggressive memory disambiguation requires no recovery since the same load instructions and their dependent instructions will be executed again un-speculatively after the prior store addresses are resolved and the speculative execution is used only for prefetching. We call this *recovery-free speculative memory disambiguation.*

The proposed changes are relatively minor and are unlikely to affect the critical path of the processor. Using the physical register file to keep the value predictions and the speculative execution results enables our approach to utilize otherwise unused machine resources and does not require additional ports to the register file.

One interesting aspect is that our scheme distinguishes value speculative execution from normal execution (using the *vp* flag). So, it only uses the results from normal execution to update the value predictor and eliminates the updates based on

previous predictions, thereby being able to achieve better prediction results (see Section 6.6.2).

Another interesting observation is that our proposed recovery-free speculative execution scheme could be viewed as a simple, yet efficient form of pre-execution. As each predicted value (or a presumably disambiguated load instruction) enables a set of dependent instructions to be executed speculatively, these speculatively executed instructions can be viewed as a pre-execution thread triggered by the prediction, though there is no explicit multi-threading support. Such pre-execution threads are dynamically constructed for each predicted value based on the data dependence relationship from the fetched instruction stream, thus taking advantage of dynamic branch prediction. Pre-execution is terminated when normal execution catches up with the pre-execution thread at the same instruction. The reason is that when the source registers of an instruction are ready, normal execution is performed and the *vp_ready* flag is not propagated anymore. The purpose of such pre-execution is to prefetch data. The pre-execution thread executes only if there are unused resources, thus avoiding resource competition with the main thread.

6.5 Experimental Methodology

We implemented the proposed technique in a detailed timing simulator using the Simplescalar [10] toolset. The underlying processor organization is based on the MIPS R10000 processor, configured as indicated in Table 6.1. In our experiments, we vary the D-cache configurations and the reorder buffer (ROB) size (or the instruction window size) of the base configuration to evaluate our proposed technique in a range of processor

models. Both computation-intensive and memory-intensive benchmarks are selected from the SPEC2000 integer benchmark suite and the Olden benchmark suite. Benchmarks *bzip2*, *gap*, *gcc*, *gzip*, and *perl* are computation-intensive and benchmarks *mcf*, *parser*, *twolf*, *health*, and *mst* are memory-intensive as they exhibit much higher data cache miss rates. The reference input data are used for the SPEC2000 benchmarks. We fast-forward 800M instructions and simulate the next 200M instructions. For the benchmark *health*, the input is '*max_level = 5 and max_time = 500*' and it runs to completion. For the benchmark *mst*, 3407 nodes are used as input and the first 2B instructions are skipped and the next 200M instructions are simulated. The baseline performance results of these benchmarks using the base processor model are shown in Table 6.2.

Table 6.1. Base processor configuration.

Instruction Cache	Size = 64 kB; Associativity = 4-way; Replacement = LRU; Line size = 16 instructions (64 bytes); Miss penalty = 10 cycles.
Data Cache	Size = 32 kB; Associativity = 2-way; Replacement = LRU; Line size = 64 bytes; Miss penalty = 10 cycles; 32 MHSRs.
Unified L2 Cache	Size = 512 kB; Associativity = 8-way; Replacement = LRU; Line size = 128 bytes; Miss penalty = 80 cycles; 64 MHSRs.
Branch Predictor	64K entry G-share; 32K entry BTB
Superscalar Core	Reorder buffer: 64 entries; Dispatch/issue/retire bandwidth: 4-way superscalar; 4 fully-symmetric function units; Data cache ports: 4
Execution Latencies	Address generation: 1 cycle; Memory access: 2 cycles (hit in data cache); Integer ALU ops = 1 cycle; Complex ops = MIPS R10000 latencies
Memory Disambiguation	Load stalls when there is a pending store with unresolved address.

Table 6.2. Baseline results of the benchmarks.

Benchmarks	Computation-Intensive					Memory-Intensive				
	bzip2	gap	gcc	gzip	perl	mcf	parser	twolf	health	mst
IPC	1.68	1.31	2.11	1.46	1.46	0.51	0.85	0.83	0.32	0.21
L1 D-cache miss rate (misses per 1K insn.)	2.14% (4.88)	0.45% (0.95)	5.29% (14.08)	6.88% (16.24)	1.98% (8.61)	46.6% (166.3)	9.12% (33.04)	14.1% (45.23)	16.3% (66.08)	55.3% (175.1)
L2 Cache miss rate (misses per 1K insn.)	28.5% (1.39)	68.3% (0.65)	46.0% (6.48)	46.6% (7.57)	40.2% (3.46)	67.5% (112.3)	48.0% (15.84)	62.2% (28.12)	85.0% (56.20)	96.4% (168.8)

As described in Section 6.4, a simple stride value predictor (tag-less 4K-entry) is used in our experiments to generate value predictions. The prediction table is indexed with the *pc* and each entry in the table has three fields, as shown in Figure 6.5. The field '*last value*' holds the most recent execution result and the field '*stride*' keeps the difference between the last two execution results. The 3-bit confidence counter is used to filter out potentially incorrect predictions. For each successful prediction, the confidence counter is increased by 2 and is decreased by 1 for each misprediction [68]. A prediction with a confidence value larger than 4 is viewed as a confident prediction. A speculative update scheme based on that proposed in [40] is also used to improve the prediction accuracy. Since there may be more than one outstanding prediction being made before any update, we use two age counters (one for prediction, one for update) per entry instead of one age counter as in [40] to keep the track of the right prediction-update pair (i.e., when an execution result is available, which prediction it corresponds to). Note that we need to perform prediction validation to update the confidence counter. This validation is

performed as part of the prediction update in the value predictor and is not part of the execution.

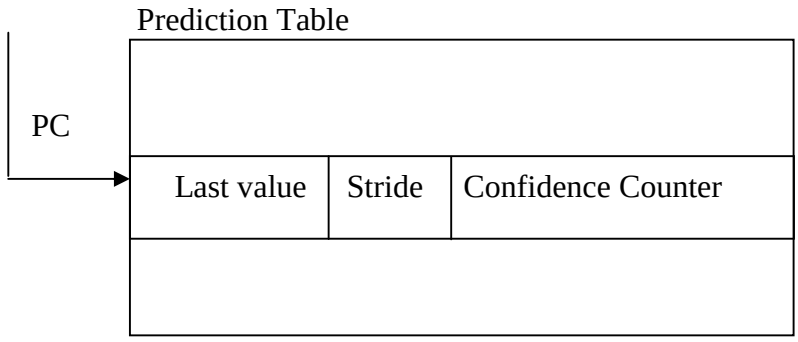


Figure 6.5. The stride value prediction table.

6.6 Experimental Results

In this section, we first evaluate the effectiveness of our proposed technique in reducing data cache miss rates, increasing MLP, and achieving performance gains. We then analyze where the performance gains come from in Section 6.6.2. In Section 6.6.3, we perform a sensitivity analysis by applying the proposed technique to a range of processor models.

6.6.1 Performance evaluation

As discussed in Section 6.4, our proposed technique breaks both true data dependencies and alias dependencies between missing loads so that the otherwise stalled loads can be executed speculatively in parallel with the un-speculative missing loads. These speculatively executed loads perform the functionality of prefetching the data into the cache so that the un-speculative execution will experience fewer cache misses. We first examine the effect of this technique in reducing data cache miss rates, as shown in

Figure 6.6 and Figure 6.7. Here, the cache misses during speculative execution are not counted since they are used to prefetch. For each benchmark in Figure 6.6, the L1 D-cache miss-rate results are reported for both the base processor (labeled '*base*') and the processor with recovery-free value prediction (labeled '*vp_exe*'). Also, the cache misses are further divided into partially covered misses (i.e., a miss request for a cache line that is already being repaired from the L2 cache or memory) and non-covered misses. Partially covered cache misses have less impact on overall performance compared to non-covered cache misses. Figure 6.6 shows that for memory intensive benchmarks, the proposed technique reduces the L1 D-cache miss rate significantly, ranging from 14% (from 47% to 33% in the benchmark *mcf*) to 0.5% (from 16.5% to 16% for the benchmark *health*) and increases the ratio of partially covered misses for most benchmarks. For computation intensive benchmarks, a visible reduction in the L1 D-cache miss rate is shown for the benchmarks *bzip2*, *gap* and *gzip* although the baseline miss-rates are relatively small for these benchmarks.

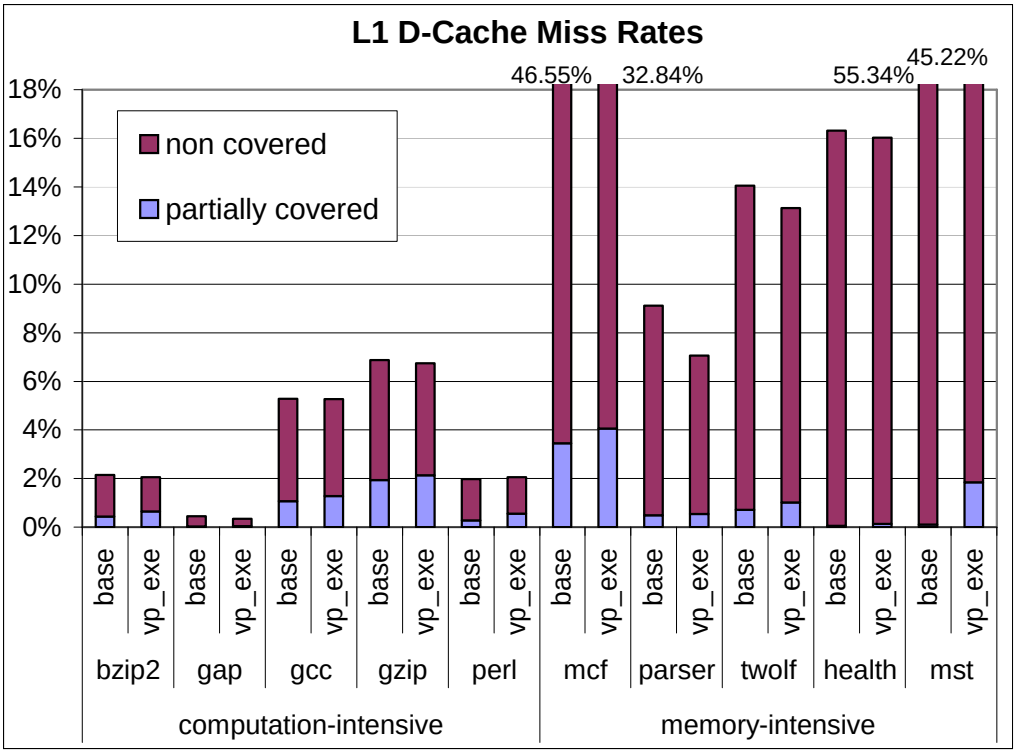


Figure 6.6. The L1 D-cache missrates.

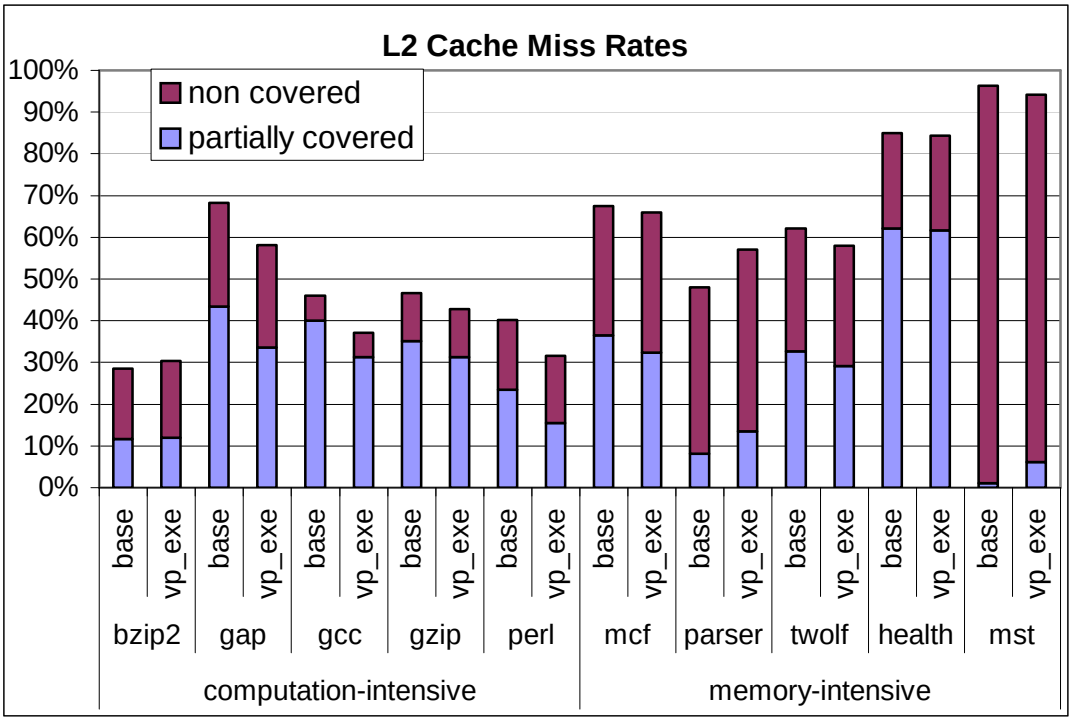


Figure 6.7. The L2 cache missrates.

Figure 6.7 shows the cache miss rate effect on the L2 caches. It can be seen that in addition to the large L1 D-cache miss rate reduction most benchmarks exhibit reduced L2 miss rates, which shows that speculative execution does not only bring the data that are already in the L2 cache into the L1 D-cache but also eliminates many L2 cache misses. For those benchmarks that exhibit increased miss rates in the L2 cache, for example the benchmark *parser*, when considering the L1 miss rate reduction, we can see that the overall L2 misses are also reduced, 14.5 L2 misses per 1k instruction compared to 15.8 L2 misses in the baseline processor.

Next, we use the benchmark *mcf* as an example to show the MLP improvement (i.e., overlapping multiple cache misses) achieved by the proposed technique for a typical heavy pointer chasing workload. Figure 6.8 shows the distribution of how many L1 data cache misses are overlapped in the base processor. The *x*-axis of Figure 6.8 is the number of overlapping misses and the *y*-axis is the time during execution that the overlapping happens. From Figure 6.8, we can see that the processor spends 12% of overall execution time on computations that do not involve a cache miss. During 33% of execution time, a single missing load is accessing the L1 D-cache (i.e., low MLP since no overlapping happens), and in 35% of the time two missing loads are accessing the L1 D-cache. The maximum number of overlapping cache misses are determined by the MSHRs used in the cache, and our experiment uses 32 MSHRs for the L1 D-cache. It can be inferred from this distribution that the benchmark *mcf* has many sequential cache misses, resulting in low MLP and MSHR utilization, and therefore long execution time.

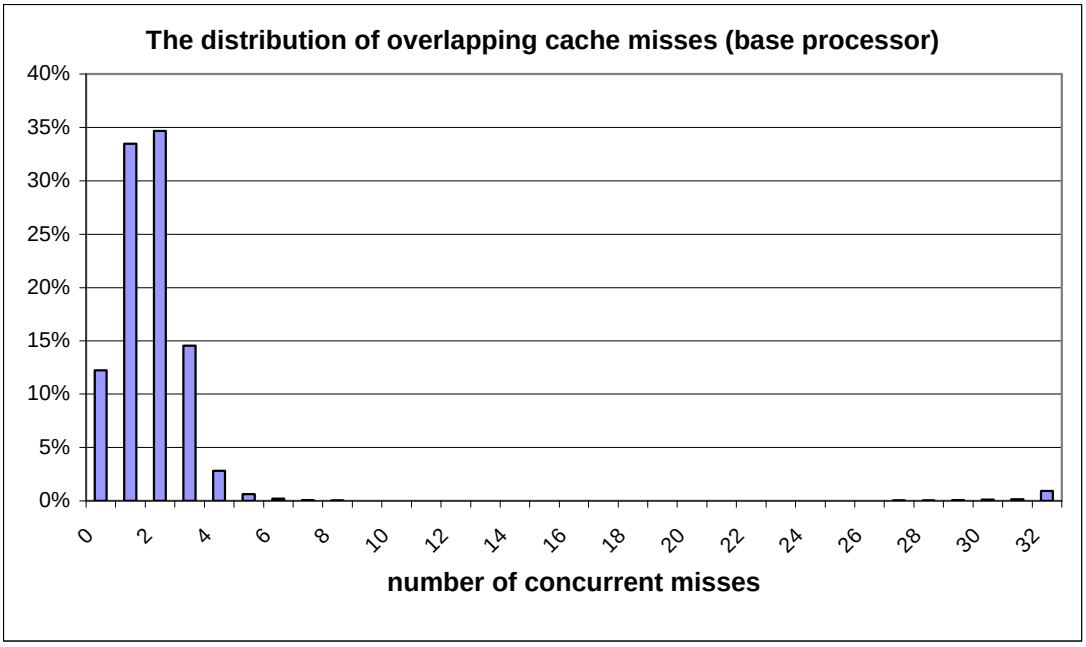


Figure 6.8. The baseline MLP for the benchmark *mcf* (overall execution time = 390M cycles).

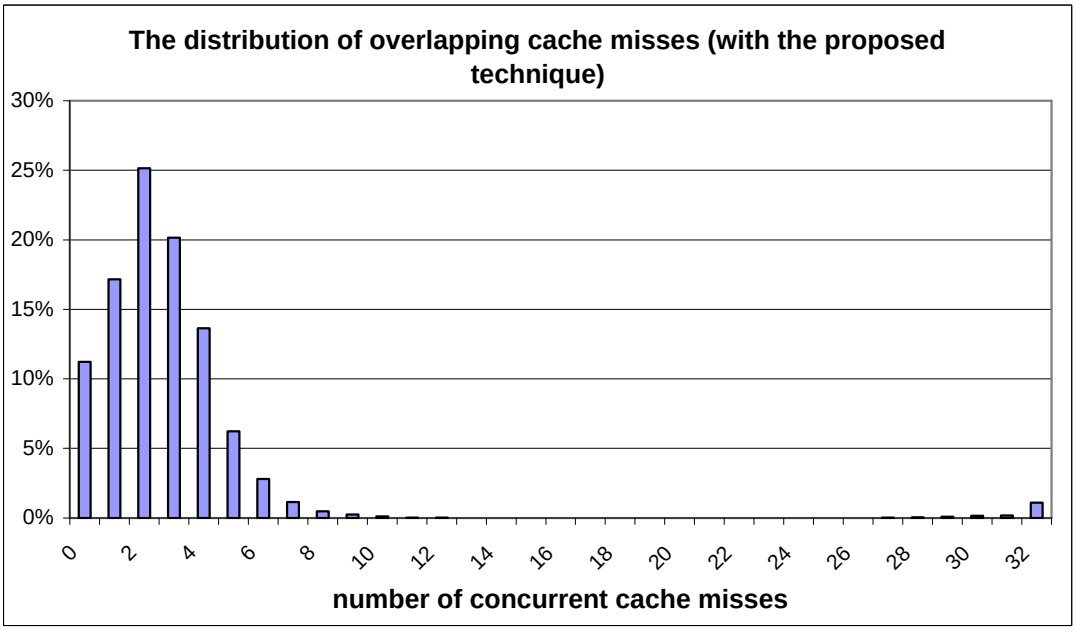


Figure 6.9. The improved MLP for the benchmark *mcf* with recovery-free value prediction (overall execution time = 327M cycles).

With recovery-free value prediction, the overall execution time is significantly reduced and MLP is much improved as shown in Figure 6.9. Compared to Figure 6.8, a significant amount of sequential cache misses are now processed in parallel. Another

interesting observation is that speculative execution does not increase the pressure on the MSHRs since it rarely converts sequential cache misses into more than six concurrent cache-misses.

Figure 6.10 shows the speedups achieved by the proposed recovery-free value prediction, and it shows that the proposed technique achieves significant speedups for memory intensive benchmarks, from 3.2% for the benchmark ***health*** to 24% for the benchmark ***mst***. For the well-known pointer-chasing benchmark ***mcf***, the speedup is 19.6%. Considering the low hardware overhead required by this technique, the performance gains are impressive. For computation intensive benchmarks, smaller speedups (average of 0.5%) result, which is expected since the reduction in the D-cache miss rate for these benchmarks is small. The only benchmark that shows a negative speedup (-0.7%) is *gcc*, which will be discussed further in Section 6.6.3.

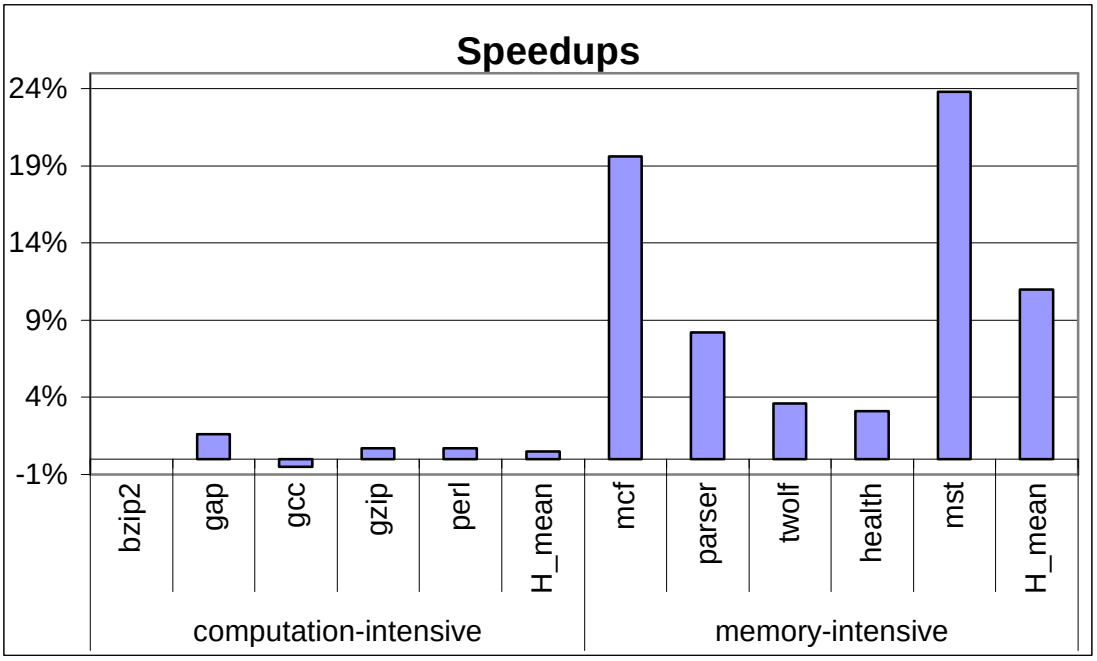


Figure 6.10. The speedups of using recovery-free value prediction.

6.6.2 Performance analysis

To analyze why the proposed technique achieves such impressive speedups, we first examine the stride value predictor to see how well it predicts a value and how often a missing load is correctly predicted.

It is observed in previous studies [21],[43],[64] that many instructions exhibit stride locality, and a more recent work [71] showed that stride locality exists in the address stream for many load instructions in irregular programs. As pointed out in Chapter 5, the predictability of load addresses is equivalent to load value predictability for pointer chasing codes. Our results, shown in Figure 6.11, confirm these observations. For each benchmark, both the value prediction coverage (i.e., the ratio of confident predictions over all predictions) and the value prediction accuracy (i.e., the ratio of the correct predictions over confident predictions) are shown in Figure 6.11 for *all* value producing instructions using a 4k-entry stride value predictor. It can be seen that most benchmarks, especially the benchmarks *mcf*, *parser*, and *mst*, exhibit a significant stride-type of value locality and this small value predictor provides decent prediction coverage and accuracy.

Since value predictions are used to break memory dependence chains, the predictability of the missing loads is of special interest and is examined in Figure 6.12. From Figure 6.12, it can be seen that the values of missing loads exhibit different degrees of stride locality for different benchmarks. For the heavy pointer chasing benchmarks *mcf* and *mst*, the value predictor achieves good prediction coverage and high accuracy. Given

their high cache miss rate and pointer chasing characteristics, this explains why these benchmarks enjoy significant speedups. For another pointer-chasing benchmark *health*, the missing loads show very limited stride-type locality. As we will see next, the speedup for this benchmark is mainly due to speculative memory disambiguation instead of breaking true memory dependencies. Again, if a more powerful predictor (e.g., a context-based predictor) is used to explore the locality in its address stream, higher speedup can be expected for this particular benchmark as well.

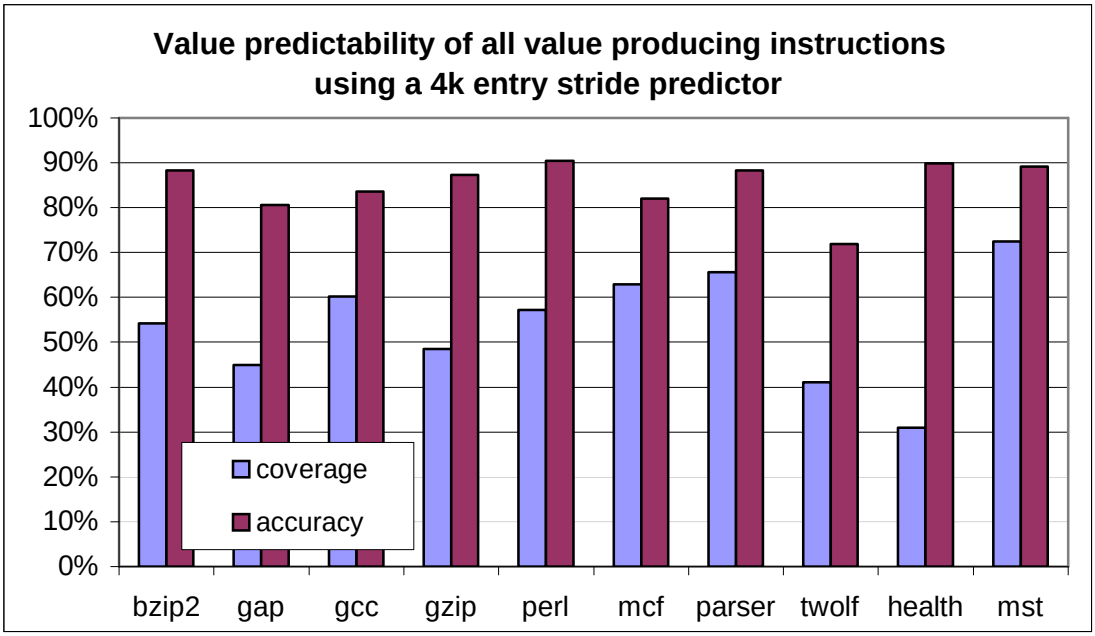


Figure 6.11. The value predictability for all value producing instructions using a 4k-entry stride predictor.

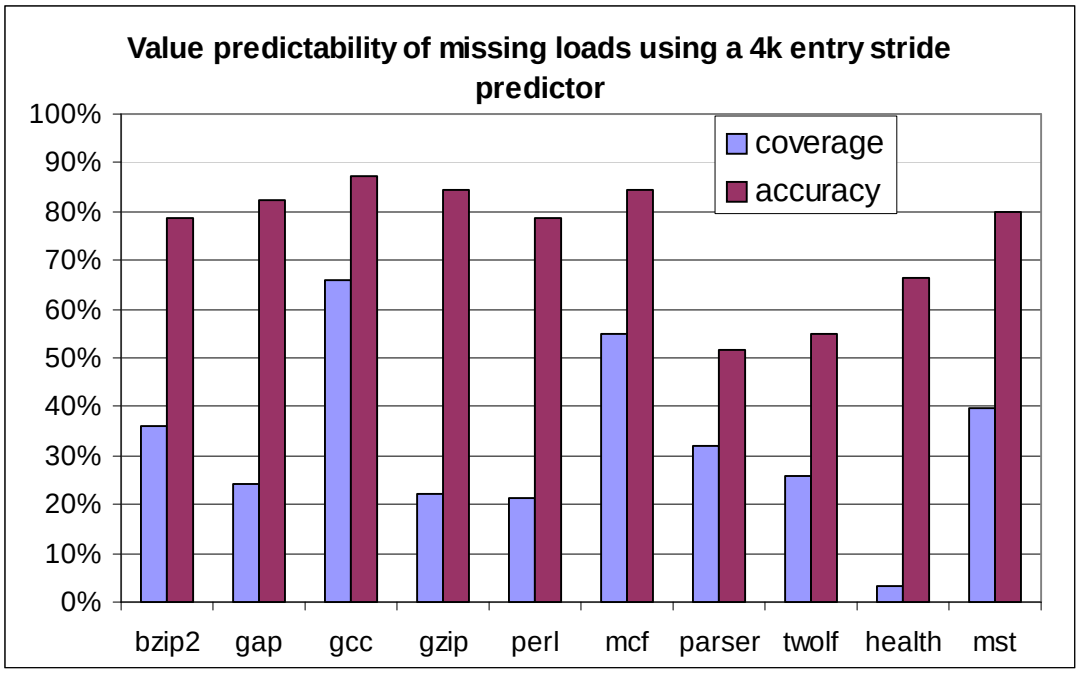


Figure 6.12 The value predictability for missing loads using a 4k-entry stride predictor.

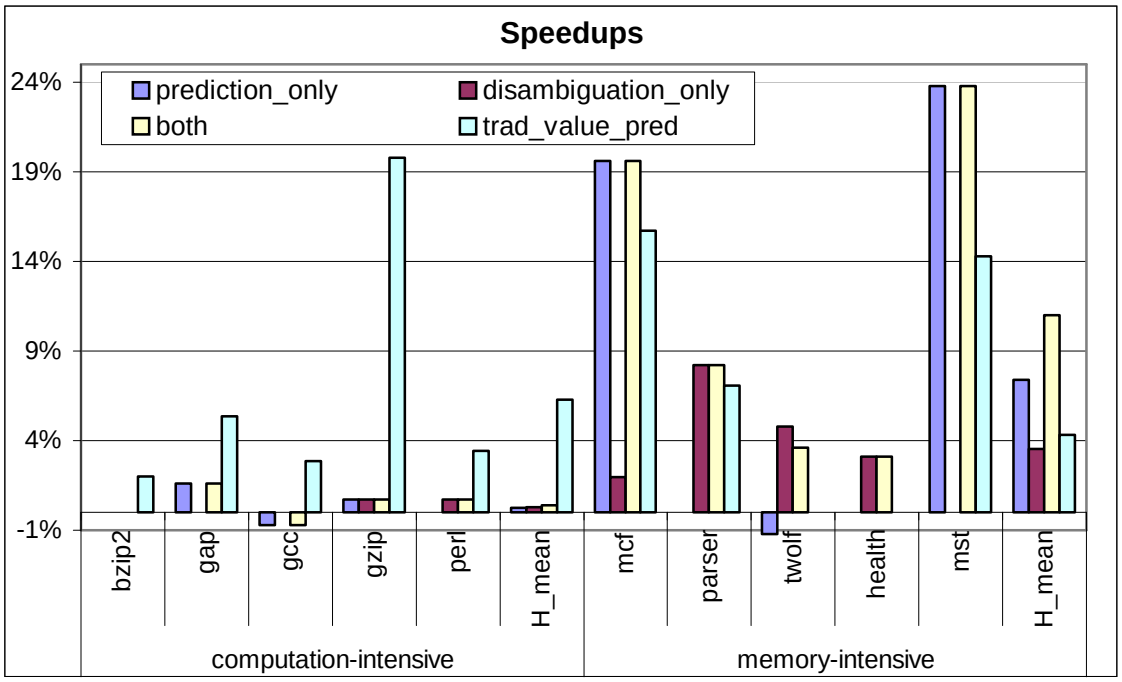


Figure 6.13. The speedups resulting from breaking different dependencies and traditional value speculation.

As discussed in Section 6.3, both true data dependence and the alias dependence between missing loads prevent them from being executed in parallel. The recovery-free value prediction scheme breaks both dependencies during speculative execution. Next, we examine the impact of breaking either of these two dependencies on enhancing MLP. In the next experiment, we isolate the performance impact by breaking only one type of dependency at a time. Figure 6.13 shows the speedup results for breaking true data dependencies only (labeled '*prediction_only*'), breaking alias dependencies only (labeled '*disambiguation_only*'), and breaking both dependencies (i.e., the same results as in Figure 6.10, labeled '*both*'). We also include the speedup results using traditional value prediction (labeled '*trad_value_pred*') in Figure 6.13. In the traditional value prediction scheme, the same stride value predictor is used and an idealistic validation and selective recovery (1 cycle penalty) mechanism is incorporated into the execution pipeline. From Figure 6.13, it can be seen that for computation-intensive benchmarks, breaking alias dependencies ('*disambiguation_only*') has slightly better speedups than breaking true dependencies only ('*prediction_only*'). For memory-intensive benchmarks, breaking true dependencies ('*prediction_only*') results in much higher speedups for *mcf* and *mst* but less speedups for other benchmarks compared to breaking alias dependencies ('*disambiguation_only*'). The reason is that for these benchmarks many memory dependence chains are formed by alias dependencies. For these benchmarks, increasing the instruction window size and performing speculative memory disambiguation can improve MLP effectively. Also, our value predictor only exploits stride locality, limiting the opportunity to break true memory dependencies more aggressively. The benchmarks *mcf* and *mst*, on the other hand, feature heavy pointer chasing and exhibit strong stride

locality in their value streams. So, breaking true dependencies becomes more profitable. Fortunately, when both true dependencies and alias dependencies are broken at the same time using our proposed approach, higher speedups are achieved. This mutually beneficial effect confirms our observation in Section 6.3 that both memory dependencies need be broken to improve MLP. Similar results are also reported in a study [11] of the interaction between value prediction and memory dependence speculation.

Comparing our proposed recovery-free scheme to traditional value prediction, we can see that traditional value prediction achieves higher speedups for computation intensive benchmarks. For memory-intensive benchmarks, our recovery-free prediction scheme has much higher speedups since it avoids the misprediction penalties and benefits from speculative memory disambiguation. For example, the recovery penalties (even with only 1 cycle penalty per misprediction) account for 2.6% of the overall execution time for the benchmark *mcf* while our recovery-free scheme completely removes such penalties. Moreover, in recovery-free value prediction, we can distinguish the speculative execution from the normal execution using the 'vp' flag. The value predictor in recovery-free value prediction is updated only with un-speculative execution results (i.e., the computation results not involving direct/indirect predicted values), thereby being able to achieve higher prediction accuracies than the traditional value speculation scheme, as seen in Figure 6.14. In Figure 6.14, the prediction coverage and accuracy for both recovery-free value prediction and traditional value prediction are shown. It can be observed that recovery-free value prediction and traditional value prediction achieve similar prediction accuracies while recovery-free value prediction exhibits better prediction coverage. To highlight these differences, we include another metric, labeled 'product', which is the

product of coverage and accuracy, representing the ratio of the number of the correct and confident predictions over the number of all the predictions produced by the value predictor. Using this metric, it can be seen that recovery-free value prediction achieves better prediction power for most benchmarks than traditional value prediction, especially for the benchmarks *mcf* and *mst*.

The results in Figure 6.13 also suggest another interesting optimization: we can apply recovery-free value prediction selectively by monitoring the dynamic behavior of a workload. Only if the workload is memory intensive (e.g., the L1 D-cache miss rate is larger than 10%), is recovery-free value prediction turned on. Otherwise, recovery-free value prediction is turned off or only the aggressive memory disambiguation feature is used for prefetching. Further exploration of this optimization is out of the scope of this chapter and left as future work.

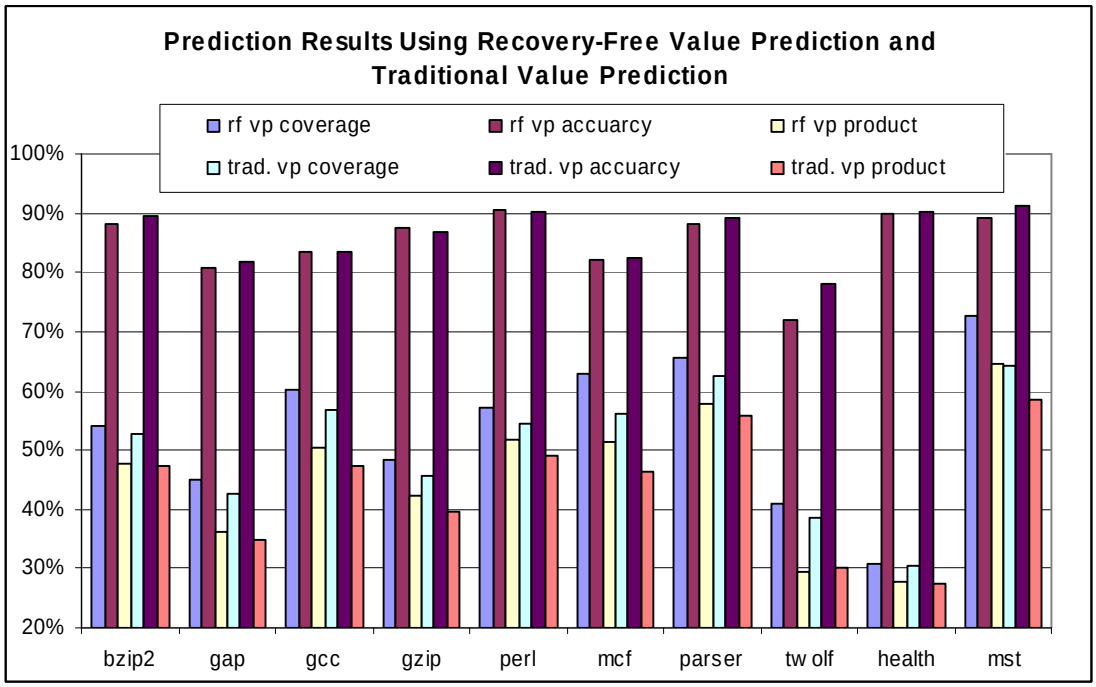


Figure 6.14. The value prediction results using recovery-free value prediction (labeled 'rf vp') and traditional value prediction (labeled 'trad. vp').

6.6.3 Sensitivity analysis

In this experiment, we evaluate our proposed technique in different memory hierarchy models, 16kB direct-mapped L1 D-cache and 256kB 4-way L2 unified cache (labeled as '*configuration 1*'), 32kB 2-way L1 D-cache and 512kB 8-way L2 cache (same as base processor, labeled as '*configuration 2*'), and 64kB 4-way L1 D-cache and 2048kB 8-way L2 cache (labeled as '*configuration 3*'). The speedups of the proposed technique in these configurations are show in Figure 6.15.

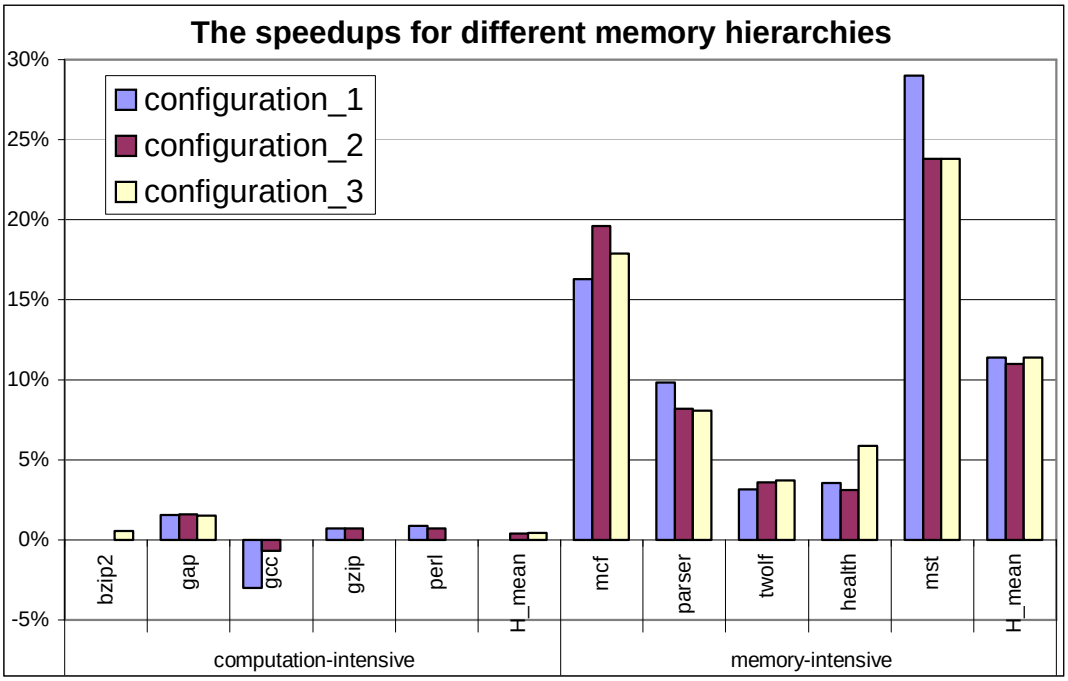


Figure 6.15. The speedups for different memory hierarchies.

Interesting observations can be made from Figure 6.15. First, for the small D-cache of 16kB, the memory problem becomes more evident. As a result, more speedups are achieved by hiding the miss latency using recovery-free value prediction, as we can see from the benchmarks, *mst* and *parser*. On the other hand, however, a small cache can tolerate less cache pollution resulting from value mispredictions. So, the miss rate can

actually increase if the value misprediction rate is high and the speedups are reduced, as in the benchmarks *gcc* and *twolf*. Large caches such as 64kB are more tolerant of cache pollution problems while the criticality of memory operations is reduced if they hit in the cache, as we can see from the benchmark *mcf* and *parser*. On average, the proposed scheme performs quite well for all the different memory hierarchy configurations.

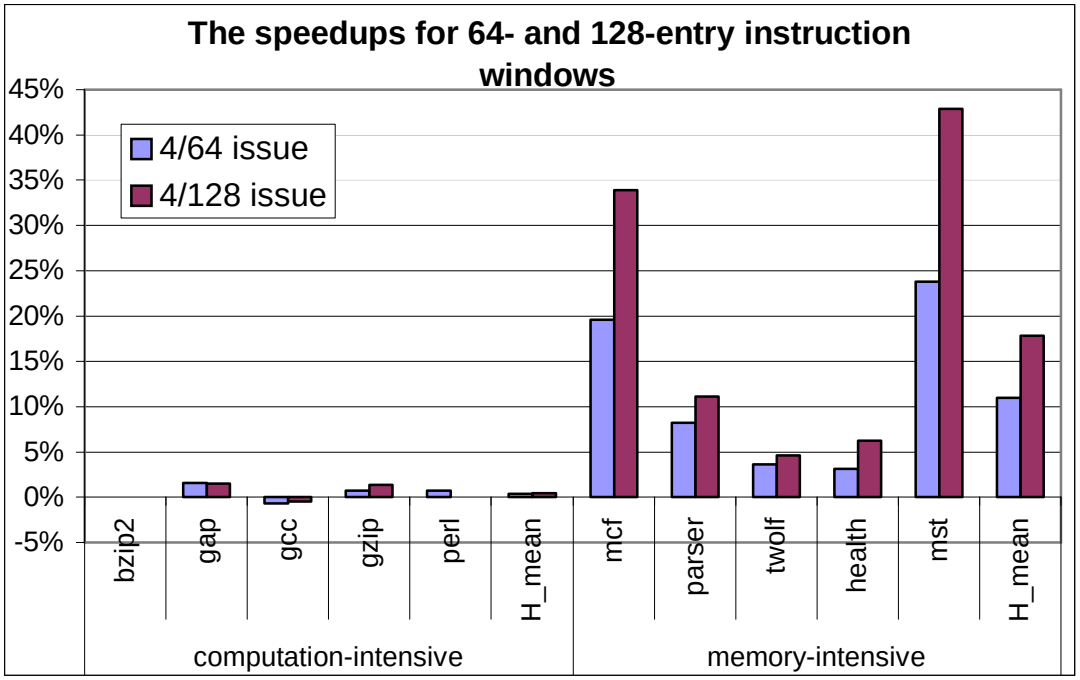


Figure 6.16. The speedups for different instruction window sizes.

In the next experiment, we increase the instruction window size to 128 to allow it to be more tolerant to L1 D-cache misses. The same 32kB 2-way L1 D-cache and 512kB 8-way L2 cache are used as in the 4/64 issue model. The results are shown in Figure 6.16. From this experiment, we can see that much higher speedups are reported for the 128-entry instruction window in all memory-intensive benchmarks using recovery-free value prediction. There are two main reasons for this trend. First, a large instruction window size of 128 holds a longer memory dependence chain. As discussed in Chapter 5, breaking a longer chain can overlap more cache misses, resulting in higher performance

improvement. Secondly, a larger instruction window enables more instructions to be fetched into the window under a long-latency cache miss, thereby enabling those instructions to be predicted earlier than in a small instruction window. As a result, speculative loads (or prefetches) can be issued earlier to hide more memory access latencies.

6.7 Limitations

Two limitations exist with our proposed scheme. First, as we pointed out in Chapter 5 and Section 6.3, value prediction can hide memory access latencies by breaking the memory dependencies, especially for long memory dependence chains. As a result, it is effective for memory-intensive workloads with heavy pointer-chasing. If a workload does not exhibit such memory dependencies, for example, the cache misses due to accessing a large array, our proposed scheme will have very limited ability to hide these cache miss penalties since the prediction of a missing load will not lead to the address of other missing loads. For those cache misses that form multiple *short* memory dependence chains, either large instruction windows [34],[39] or the address prediction based memory prefetching would be more effective, as we discussed in Chapter 5.

Secondly, in our proposed recovery-free value prediction scheme, a prediction is made only after the instruction is fetched, and the prediction is consumed only when the dependent instructions are in the instruction window. This implies that the earliest time for a speculative load to be executed is after the load instruction is dispatched into the instruction window. It limits the capability to explore the far-flung MLP even when the correct prediction can be made. Experiments in Section 6.3 show the performance impact

of using a large instruction window to bring in instructions early into the instruction window. Another interesting way to explore the distant MLP is to combine our approach with run-ahead execution [18],[57] to pre-execute/prefetch both independent and dependent memory accesses.

6.8 Summary

In this chapter, we advocate using value prediction to enhance MLP for memory intensive benchmarks with heavy pointer chasing. As current microprocessors can execute instructions very fast as long as long memory latency operations, such as cache misses, are not involved, we propose to use value prediction only for data prefetching so that complex prediction validation and misprediction recovery mechanisms are avoided and only minor hardware changes are necessary. Also, the same hardware changes enable aggressive memory disambiguation for prefetching.

We present our design of recovery-free value prediction based on a MIPS R10000 processor model, and the simulation results show that our technique enhances MLP effectively for a range of memory-intensive benchmarks and achieves significant speedups.

As pointed out in [1], only a few static load instructions are responsible for the majority of dynamic cache misses. So, it would be very interesting to tune the value predictor to predict only the values leading to the address computation of these load instructions. This would further reduce the hardware overhead and the power consumption overhead due to useless speculation (i.e., the speculation not leading to useful prefetch). Also, in our implementation of recovery-free value prediction,

speculative loads are treated the same way as normal loads though their purpose is to prefetch. So, one way to reduce the cache pollution effect is to store the prefetched data block in the LRU entry and inherit the LRU instead of treating the data as MRU.

In the current implementation of recovery-free value prediction, we prohibit the speculative execution of store and branch instructions. Previous studies [25],[29] show that value prediction can also be used to improve branch prediction results. So, one interesting way to extend recovery-free value prediction is to selectively perform branches during the speculative execution to explore the control speculation effect.

Chapter 7 Conclusion and Future Directions

In this dissertation, we investigate both compiler and microarchitecture design techniques to achieve performance improvement. In order to evaluate the performance impact efficiently, a set of performance bounds are proposed based on different workload characteristics and different target microarchitectures.

For ILP dominated workloads, we propose a low complexity, bound-guided approach to systematically regulate code size related ILP optimizations during code compilation. Such a bound-guided approach captures the performance impact as well as the overhead in static code size increase of an optimization using a concept called code size efficiency. Based on their efficiencies, the ILP optimizations are performed selectively so that performance is highly improved at a minor cost in static code size increase. The ‘90/10’ rule and the dependence height reduction impact, embodied in the definition of code size efficiency, result in a very interesting diminishing return phenomenon. Based on this phenomenon, we define an optimal trade-off between the ILP improvement and the static code size increase and develop a very simple threshold scheme to achieve this optimum. The experimental results using the SPEC 2000 INT

benchmark suite validate our proposed techniques and show significant speedups with little code size increase from the selectively performed ILP optimizations. A similar approach is also developed for real-time systems to reduce the WCET effectively.

For memory-intensive workloads, our focus is to improve MLP as memory accesses consume the majority of the overall execution time. We first perform performance modeling using performance bounds to evaluate two well-known latency hiding techniques. With the key insight revealed from the modeling, we propose a cost-effective approach, namely recovery-free value prediction, to enhance MLP for memory intensive workloads with heavy pointer chasing. In this scheme, both true memory dependency and alias dependency are broken speculatively and the speculation is only used for prefetching. As a result, such speculation achieves an effect similar to pre-execution to warm up the data cache, and therefore is recovery-free. The experiments show that the proposed technique improves MLP effectively and achieves impressive speedups for the target memory–intensive workloads.

The work in this dissertation can be extended in following directions.

- The profile guided performance bounds rely on accurate edge profiling information to reveal which part of the program is most frequently executed (or hot spots). It would be very interesting to investigate how much the performance gains achieved using our proposed scheme are affected by profile variation. Also, it is interesting to see whether and how our proposed scheme benefits from techniques such as dynamic

compilation and hot spot identification to exploit the more accurate timing profile information.

- Performance bounds are a very useful concept. We use them in computing the code size efficiency to achieve a good tradeoff between performance improvement and static code size increase. A similar idea can be used for other design trade-off evaluations. One good direction is power/energy consumption and the performance improvement due to different kinds of speculation.
- In our study of compilation for real-time applications, we use selected benchmarks from the SPEC 2000 INT suite and the MiBench suite. However, these benchmarks are not designed for real-time application purposes, e.g., there is no information on execution deadlines. There are benchmarks developed for static WCET analysis, such as C-Lab [80]. But, these benchmarks are very simple in terms of code structures and workload characteristics. Considering the current trend in real-time processing, such as video and audio processing, a more complete benchmark suite based on practical workloads would benefit future work in this area.
- Our analytical model in Chapter 5 showed that prefetching is more effective in short memory dependence chains while value prediction has better potential in long memory dependence chains. It would be very interesting to integrate these two schemes, using either the compiler or some adaptive hardware to further improve memory latency hiding.

- Recovery-free value prediction has an effect similar to pre-executing dependent instructions of the cache-missing loads while runahead execution pre-executes independent instructions to warm the caches. Promising results can be expected from combining these two techniques.
- Currently proposed recovery-free value prediction has low hardware overhead but has relatively large overhead in terms of speculatively executed instructions. As pointed out before, a few static load instructions are responsible for a majority of dynamic cache misses. We can tune the value predictor to predict only those values that lead to the address computation of these static loads. Such selective recovery-free value prediction also reduces the hardware overhead due to the value prediction table.
- In addition to the memory wall problem, control dependence presents another great challenge for current microprocessor design. As value prediction is shown to be effective in improving branch prediction results, a very interesting extension to recovery-free value prediction is to explore its effect on control speculation.

Chapter 8 Bibliography

[1] S. G. Abraham, R. A. Sugumar, D. Windheiser, B. R. Rau, and R. Gupta, "Predictability of load/store latencies", *Proceeding of the 26th International Symposium on Microarchitecture (MICRO-26)*, 1993.

[2] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence", *Proceeding of 10th ACM Symposium on Principles of Programming Languages*, 1983.

[3] A. Anantaraman, K. Seth, K. Patil, E. Rotenberg, and F. Mueller. "Virtual Simple Architecture (VISA): Exceeding the Complexity Limit in Safe Real-Time Systems". *Proceeding of the 30th International Symposium on Computer Architecture (ISCA-30)*, June 2003.

[4] D. I. August, W. W. Hwu, and S. A. Mahlke, "A framework for balancing control flow and predication", *Proc. 30th Ann. Int'l Symp. Microarchitecture (MICRO30)*, 1997.

[5] M. Bekerman, S. Jourdan, R. Ronen, G Kirshenboim, L. Pappoport, A. Yoaz, and U. Weiser, "Correlated Load-Address Predictors", *Proceeding of the 26th International Symposium on Computer Architecture (ISCA-26)*, 1999.

[6] D. Bernstein, D. Cohen, and H. Krawczyk, "Code Duplication: An Assist for Global Instruction Scheduling", *Proc. 24th Ann. Int'l Symp. Microarchitecture (MICRO24)*, 1991.

[7] J. Bharadwaj, K. Menezes, and C. McKinsey, "Wavefront scheduling: path based data representation and scheduling of subgraphs", *Proc. 32nd Ann. Int'l Symp. Microarchitecture (MICRO32),* 1999.

[8] E. L. Boyd, "Performance evaluation and improvement of parallel applications on high performance architectures", *Ph.D. thesis, University of Michigan*, 1995.

[9] E. Boyd, W. Azeem, H. Lee, T. Shih, S. Hung, and E. Davison, "A hierarchical pproach to modeling and improving the performance of scientific applications on the KSR1", *Proc. Of the 1994 Int'l. Conf. On Parallel Processing*, 1994.

[10] D. Burger and T. Austin, "The SimpleScalar tool set, v2.0", *Computer Architecture News (ACM SIGARCH newsletter),* vol. 25, June 1997.

[11] B. Calder and G. Reinman, "A comparative survey of load speculation architecures", *Journal of Instruction-Level Parallelism*, 2000.

[12] M. Carlisle, "Olden: parallelizing programs with dynamic data structures on distributed-memory machines", *Ph.D. thesis, Princeton University Computer Science Department,* 1996.

[13] K. Chen, S. Malik, and D. August, "Retargetable static timing analysis for embedded software", *Int'l Symp. on System Synthesis (ISSS'01),* 2001.

[14] T. F. Chen and J. L. Baer, "Reducing memory latency via non-blocking and prefetching caches", In *Proc. of the 5th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 1992.

[15] Y. Choi, A. Knies, L. Gerke, and T. Ngai, "The impact of If-conversion and branch prediction on program execution on the Intel Itanium processor", *Proc. 34th Ann. Int'l Symp. Microarchitecture (MICRO34),* 2001.

[16] J. D. Collins, H. Wang, D. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen, "Speculative precomputation: long-range prefetching of delinquent loads", *Proceeding of the 28th International Symposium on Computer Architecture (ISCA-28),* 2001

[17] R. Cooksey, S. Jourdan, and D. Grunwald, "A stateless, content-directed data prefetching mechanism", Proceeding of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X), 2002.

[18] J. Dundas, and T. Mudge, "Improving data cache performance by pre-executing instructions under a cache miss", *Proceeding of the 1997 International Conference on Supercomputing*, 1997.

[19] A. E. Eichenberger and W. M. Meleis, "Balance Scheduling: Weighting Branch Tradeoffs in Superblocks", *Proc. 32nd Ann. Int'l Symp. Microarchitecture (MICRO32)*, 1999.

[20] C. Fu, "Compiler driven value speculation scheduling", *Ph.D. thesis*. ECE Department, N. C. State University, 2001.

[21] F. Gabbay and A. Mendelson, "Speculative execution based on value prediction," *EE Department Tech Report 1080, Tachnion - Israel Institute of Technology*, Nov. 1996.

[22] R. Gerber and S. Hong, "Compiling real-time programs with timing constraint refinement and structural code motion", *IEEE Trans. on Software Engineering*, Vol. 21, No. 5, May 1995.

[23] R. Gerber and S. Hong, "Slicing real-time programs for enhanced schedulability", *ACM Trans. on Programming Language and Systems*, Vol. 19, No. 3, 1997.

[24] J. Gonzalez and A. Gonzalez, "Speculative execution via address prediction and data prefetching", *Proceeding of the 1997 International Conference on Supercomputing*, 1997.

[25] J. Gonzalez and A. Gonzalez, "Control-Flow Speculation through Value Prediction for Superscalar Processors", *Proc. of the 1999 Conf. On Parallel Architectures and Compilation Techniques (PACT'99)*, 1999.

[26] P. Gopinath and R. Gupta, "Applying compiler techniques to scheduling in real-time systems", *Proc. of the 11th IEEE Real-Time Systems Symposium (RTSS)*, 1990.

[27] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, R. Brown, "MiBench: a free, commercially representative embedded benchmark suite", *2001 IEEE Int'l Workshop on Workload Characterization (WWC-4)*, 2001.

[28] 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 (HPCA-4)*, February 1998.

[29] T. Heil, Z. Smith, and J. E. Smith, "Improving branch predictors by correlating on data values", in *32nd International Symposium on Microarchitecture* (MICRO-32), 1999.

[30] J. Henning, "SPEC2000: measuring CPU performance in the new millennium", *IEEE Computer*, July 2000.

[31] 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 way for VLIW and superblock compilation." *The Journal of Supercomputing*, vol. 7, pp. 229-248, January 1993.

[32] Intel Corp, IA-64 Application Developer's Architecture Guide, 2000.

[33] D. Joseph and D. Grunwald, "Prefetching using Markov Predictors", *IEEE Transactions on Computers.* Vol. 48, Feb 1999.

[34] T. Karkhanis and J. Smith, "A Day in the Life of a Cache Miss", Proceeding of the 2nd Annual Workshop on Memory Performance Issues (WMPI 2002), 2002.

[35] V. Kathail, M. S. Schlansker, and B. R. Rau, "HPL-PD architecture specification: version 1.1." *Tech. Rep. HPL-93-80 (R.1), Hewlett--Packard Laboratories*, February 2000.

[36] T. Kisuki, P.M.W. Knijnenburg, and M.F.P. O'Boyle, "Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation", *Proc. of the 2000 Conf. On Parallel Architectures and Compilation Techniques (PACT'00)*, October 2000.

[37] M. Langevin and E. Cerny, "A recursive technique for computing lower bound performance of schedules", *IEEE International Conference on Computer Design (ICCD)*, 1993.

[38] D. M. Lavery and W W. Hwu, "Unrolling-based optimizations for modulo scheduling", *Proc. 28th Ann. Int'l Symp. Microarchitecture (MICRO28)*, 1995.

[39] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg, "A large, fast instruction window for tolerating cache misses", *Proceeding of the 29th International Symposium on Computer Architecture (ISCA-29)*, 2002.

[40] S. Lee and P. Yew, "On some implementation issues for value prediction on wide ILP processors", Proceeding of the International Conference on Parallel Architectures and Compilation Techniques (PACT'00), 2000.

[41] The LEGO Compiler. Available for download at http://www.tinker.ncsu.edu/LEGO.

[42] A. Leung, K. Palem, and A. Pnueli, "A fast algorithm for scheduling time-contrained instructions on processors with ILP", *Proc. Of the 1998 Conf. On Parallel Architectures and Compilation Techniques (PACT'98)*, 1998.

[43] M. H. Lipasti and J. P. Shen, "Exceeding the dataflow limit via value prediction," *Proceeding of the 29th International Symposium on Microarchitecture (MICRO-29)*, 1996.

[44] M.H. Lipasti, C. B. Wikerson and J. P. Shen, "Value locality and load value prediction," Proceeding of the 7th International Conference on Architectural Support for Programming Language and Operation Systems (ASPLOS-7), Oct, 1996.

[45] J. W. S. Liu, *Real-Time Systems*, Prentice Hall, Upper Saddle River, New Jersey, 2000.

[46] J. W. S. Liu, K. J. Lin, C. L. Liu, and C.W. Gear, "Research on Imprecise Computations in Project Quartz", *Proceedings of the 1989 Workshop on Operating Systems for Mission Critical Computing*, 1989.

[47] C. K. Luk, "Tolerating memory latency through soft-ware-controlled pre-execution in simultaneous multithreading processors", *Proceeding of the 28th International Symposium on Computer Architecture (ISCA-28)*, 2001.

[48] T. Lundqvist and P. Stenstrom, "An integrated path and timing analysis method based on cycle-level symbolic execution", *Real-Time Systems*, 17(2/3): 183-207, 1999.

[49] S. A. Mahlke, "Exploiting instruction level parallelism in the presence of branches", *PhD thesis*, ECE Department, Univ. of Illinois at Urbana-Champaign, 1996.

[50] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher, and W. W. Hwu, "Characterizing the impact of predicated execution on branch prediction", *Proc. 27th Ann. Int'l Symp. Microarchitecture (MICRO27)*, 1994.

[51] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann "Effective compiler support for predicated execution using the Hyperblock" *Proc. 25th Ann. Int'l Symp. Microarchitecture (MICRO25)*, December 1992.

[52] W. Mangione-Smith, “Performance.bounds and buffer space requirements for concurrent processors”, *Ph.D. thesis, University of Michigan*, 1992.

[53] Bill Mangione-Smith, “Performance Bounds for Rapid Computer System Evaluation”, *Fast Simulation of Computer Architectures, edited by Thomas M. Conte and Charles E. Gimarc*, Kluwer Academic Publishers, 1995.

[54] W. Mangione-Smith, S. Abraham, and E. Davison, “The effects of memory latency and fine-grain parallelism on Astronautics ZS-1 performance”, *Proc. of the 23rd Annual Hawaii Int’l Conf on System Science*, 1990.

[55] S. Mantripragada and A. Nicolau, “Using profiling to reduce branch misprediction costs on a dynamically scheduled processor”, *Proceedings of International Conference on Supercomputing (ICS)*, 2000.

[56] F. Mueller and D. B. Whalley, "Avoiding Conditional Branches via Code Replication", *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-1995)*, June 1995.

[57] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt, “Runahead execution: an alternative to very large instruction windows for out-of-order processors”, *Proceeding of the 9th International Symposium on High Performance Computer Architecture (HPCA-9)*, 2003.

[58] J. C. Park and M. S. Schlansker, “On predicated execution”, Tech. Rep. HPL-91-58, Hewlett--Packard Laboratories, 1991.

[59] B. R. Rau, “Iterative Module Scheduling”, Tech. Rep. HPL-94-115, Hewlett-Packard Laboratories, 1995.

[60] M. Rim and R. Jain, “Lower-bound performance estimation for high-level synthesis scheduling problem”, *IEEE Trans. on CAD of Integrated Circuits and Systems*, 13(4), 1994.

[61] E. Rotenberg, S. Bennett, and J. E. Smith. "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching", *Proceeding of the 29th International Symposium on Microarchitecture (MICRO-29)*, 1996.

[62] A. Roth and G. Sohi, “Speculative data driven multithreading”, Proceeding of the 7th International Symposium on High Performance Computer Architecture (HPCA-7), 2001.

[63] V. Sarkar, “Optimized Unrolling of Nested Loops”, Proceedings of International Conference on Supercomputing (ICS), 2000.

[64] Y. Sazeides and J. E. Smith, “The predictability of data values,” *Proceeding of the 30th International Symposium on Microarchitecture (MICRO-30)*, Nov. 1997.

[65] M. S. Schlansker and B. R. Rau. “EPIC: An architecture for instruction-level parallel processors” *Tech. Rep. HPL-99-111, Hewlett--Packard Laboratories*, February 2000.

[66] E. Sprangle and D. Carmean, “Increasing processor performance by implementing deeper pipelines”, *Proceedings of the 29th International Symposium on Computer Architecture (ISCA-29)*, 2002.

[67] J. Tang and E. Davison, “An evaluation of Cray-1 and Cray-X-MP performance on vectorizable Livermore Fortran kernels”, *Proc. of Int’l Conf. On Supercomputing*, July 1988.

[68] K. Wang and M. Franklin, “Highly accurate data value prediction using hybrid predictors,” *Proceeding of the 30th International Symposium on Microarchitecture (MICRO-30)*, Nov. 1997.

[69] P. H. Wang, H. Wang, J. D. Collins, E. Grochowski, R. M. Kling, and J. P. Shen, “Memory latency-tolerance approaches for Itanium processors: out-of-order execution vs. speculative precomputation”, *Proceeding of the 8th International Symposium on High Performance Computer Architecture (HPCA-8)*, 2002.

[70] H. Wu and J. Joxan, “An efficient algorithm for scheduling instructions with deadline constraints on ILP processors”, *Proc. of the 22nd IEEE Real-Time Systems Symposium (RTSS)*, 2001.

[71] Y. Wu, “Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching”, *Proceeding of the ACM 2002 Conference on Programming Language Design and Implementation (PLDI-2002)*, 2002.

[72] Y. Wu and J. Larus, “Static branch frequency and program profile analysis”, *Proc. 27th Ann. Int’l Symp. Microarchitecture (MICRO27)*, 1994.

[73] K. C. Yeager, “The MIPS R10000 superscalar microprocessor”, *IEEE Micro*, 1996.

[74] H. Zhou, J. Bodine, and T. Conte, "Detecting global stride localities in value streams", *Proceeding of the 30th International Symposium on Computer Architecture (ISCA-30)*, 2003.

[75] H. Zhou and T. M. Conte, "Code size efficiency in global scheduling for ILP processors", *6th workshop on Interaction between Compilers and Computer Architecture (INTERACT-6)*, Feb. 2002.

[76] H. Zhou and T. M. Conte, "Enhancing memory level parallelism via recovery-free value prediction", *Proceedings of 2003 International Conference on Supercomputing (ICS)*, 2003.

[77] H. Zhou, C. Fu, E. Rotenberg, and T. Conte, "A study of value speculative execution and misspeculation recovery in superscalar microprocessors", *Technical Report, ECE Department, N. C. State University*, Jan., 2001.

[78] H. Zhou, M. Jennings, and T. M. Conte. "Tree Traversal Scheduling: A Global Scheduling Technique for VLIW/EPIC Processors". *Proceedings of the* 14th *Annual Workshop on Languages and Compilers for Parallel Computing (LCPC'01), LNCS*, Springer Verlag, 2001.

[79] C. Zilles and G. Sohi, "Execution-based prediction using speculative slices", *Proceeding of the 28th International Symposium on Computer Architecture (ISCA-28)*, 2001.

[80] "C-lab: WCET benchmarks", http://www.c-lab.de.

Appendix A

The first derivative of the IPC over relative code size increase ratio can be derived as following:

$$\frac{dIPC_{static}}{dSize_{relative}} = \frac{d\left(IC_{dynamic}/LBET\right)}{dSize_{relative}} = \frac{-\left(IC_{dynamic}/LBET^{2}\right)dLBET}{d\left(Size_{absolute}/IC_{static}\right)} = \frac{IPC_{static}}{\left(LBET/IC_{static}\right)} * \frac{d(-LBET)}{dSize_{absolute}} \quad \textbf{Equation a-1}$$

where the term $IC_{dynamic}$ is the effective dynamic (retiring) operation count of the program and it remains unchanged in spite of further code optimizations. The term *LBET* is the lower bound of execution time of the program level. The ratio of these two terms is IPC_{static} representing the ILP features of the original program. The term IC_{static} represents original program size in terms of the operation count and $size_{absolute}$ is the program size in terms of the operation count after performing code size increase optimizations. So, the term $dSize_{absolute}$ represents the static code size increase due to those optimizations.

If we want to set the threshold as $dIPC/dSize_{relative} \geq K$, we then have:

$$\frac{d(-LBET)}{dSize_{absolute}} \geq \frac{K * LBET}{IPC_{static} * IC_{static}} \quad \textbf{Equation a-2}$$

which is the same as Equation 3-4. Here, we use the ratio of absolute IPC changes over relative code size changes as the code size efficiency. If we want to use the ratio of the relative IPC change (i.e., the speedup) over relative code size increase as the efficiency, the IPC_{static} factor will disappear in Equations a-1 and a-2.