MEMORY PROFILING FOR DIRECTING DATA SPECULATIVE OPTIMIZATIONS AND SCHEDULING

BY

DANIEL ALEXANDER CONNORS

B.S., Purdue University, 1994

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1997

Urbana, Illinois

ACKNOWLEDGEMENTS

I would first like to thank my advisor, Professor Wen-mei Hwu, for giving me the opportunity to pursue my goals in computer architecture research. His integrity as an outstanding teacher has influenced my graduate career more than any other teacher.

My appreciation and thanks extend to many of the members of the IMPACT group. Among them, I would like to thank Rick Hank, my mentor, for giving me valuable advice concerning many things. For their help in the development of my thesis and my research, I would like to thank John Gyllenhaal, Dan Lavery, Scott Mahlke, and Teresa Johnson. In addition, David August was instrumental to the writing and development of this thesis.

I am especially grateful for my family and all my friends. Jaymie Braun's enthusiasm made my life in graduate school more enjoyable, while Margaret Carns' southern charm brightened my outlook on life. My brother Mel's sense of humor and caring nature helped me more than anyone on Earth. Also I would like to thank my sister Melody for being my biggest fan and giving me all of her attention. Finally and most importantly, I extend my love and appreciation to my mother. She is by far the most supportive person in my life.

TABLE OF CONTENTS

1. INTRODUCTION

Speculative execution is a powerful technique for exploring the full potential of wide-issue superscalar and very long instruction word (VLIW) processors. There are two classes of speculative execution, control speculation and data speculation. Control speculation refers to the execution of operations before it has been determined that they would be executed in the normal control flow of execution. The concept of control speculation has long been recognized for its ability to improve code performance [1]. Data speculation involves executing load instructions before possibly aliased operations such as stores. Although similar in nature to control speculation, the concepts concerning data speculation are just now maturing. In this thesis, a research infrastructure is presented for supporting data speculative optimizations and scheduling.

Data speculation attempts to overcome the existence of ambiguous memory dependences which research [1] has indicated as being a major impediment to exploiting instruction-level parallelism (ILP). Ambiguous memory dependences occur when instructions may possibly, but not certainly, reference the same memory location. Both the loss of source-level information and naturally occurring program aliases can create these dependences within a compiler's internal representation of a program.

One approach to overcoming the effect of memory aliasing within a program is to perform memory dependence analysis on the source-level code. The resulting analysis information can then be transferred to the intermediate representation and used to construct the true dependences. This approach is referred to as a Memory Dependence Tracking System (MDTS). The Illinois Microarchitecture Project Utilizing Advanced Compiler Technology (IMPACT) compiler uses a MDTS system known as *Sync Arcs* [2]. The purpose of such systems is to provide additional information to optimization and scheduling routines. However, an MDTS alone does not facilitate data speculation within a program.

A second approach to solving the memory aliasing problem evaluates dependences at run time, thus allowing data speculative execution. The most common method uses an architectural structure called the Memory Conflict Buffer (MCB) [3]. This method uses a buffer to record the speculative load address and its register destination. When subsequent store operations occur, the store address is compared with the load address entries within the conflict buffer. A conflict between addresses is caught by the MCB and signaled by a check operation placed at the original location of the load operation.

When speculative loads conflict frequently, data speculation will result in costly correction code, effectively degrading performance. As such, it is important for a compiler to have the ability to determine when data speculation is profitable. In fact, opportunities for data speculation within traditional optimizations such as loop invariant code removal further strengthen the need for such capabilities. Likewise, advanced instruction scheduling schemes such as superblocks [4] and hyperblocks [1] would definitely benefit from productive data speculation. Profiling is one method for estimating optimization profitability. The use of profile information in relation to control flow based optimization has been studied previously [5]. These results illustrate the benefit of using profile information within compiler optimizations. Similarly, memory profiling can provide a method of triggering data speculation in both the compiler's optimization and scheduling domains.

This thesis will illustrate how memory reference information is used within the IMPACT compiler to initiate data speculation. Then several approaches for collecting conflict information between memory operations are identified. Finally, the benefit of memory profiling within data speculative algorithms will be compared with existing algorithms.

Chapter 2 describes the memory aliasing and analysis problem, while also providing background to the Memory Conflict Buffer (MCB) data speculation approach. This chapter also presents an overview of the IMPACT compiler used throughout this thesis. Chapter 3 provides a detailed discussion of several data speculative optimizations and their associated algorithms. Chapter 4 introduces the problem of gathering effective conflict information and presents several different memory profiling approaches. Next, Chapter 5 illustrates the resulting performance of combined data speculation and memory dependence profiling. Finally, Chapter 6 contains conclusions and a discussion of future work.

2. BACKGROUND

Memory-dependence profiling interacts with several key phases of an advanced compiler, and thus warrants some background information. For instance, the accuracy of any existing dependence analysis framework directly relates to the function that memory profiling can perform within the compiler. Likewise, it is important to understand how memory dependence profiling intends to aid the data speculation model or the instruction scheduling method. For these reasons, an overview of the IMPACT research framework will be discussed. The framework consists of two separate parts, the IMPACT compiler and the IMPACT simulator. In addition to the overview of IMPACT, the necessary background on the MCB approach to data speculation is presented.

2.1 Overview of the IMPACT Compiler

The IMPACT compiler has been recognized for its ability to effectively utilize profile information within speculative and predicated execution compilation techniques. As such, it provides an excellent framework for investigating memory profiling and its application to data speculation. A block diagram of the IMPACT compiler is presented in Figure 2.1. The compiler is divided into two parts, *Pcode* and *Lcode*. *Pcode* is at the level closest to source code. Within *Pcode*, the IMPACT compiler performs loop transformations [6], function inlinning [7], profiling, and dependence analysis [2],[8]. The *Pcode* techniques for performing memory dependence analysis will be explained further in Section 2.3 of this chapter.

Lcode is the lower level of program representation, which can be viewed as an instruction set for a virtual load/store architecture. All machine independent optimizations [9] are applied at this level. Likewise, advanced compilation techniques such as superblock and hyperblock formation are also performed on the *Lcode* representation of programs.

The machine independent nature of the *Lcode* format has facilitated the creation of several code generators for different architectures. The most actively supported architectures are the Sun SPARC, the HP PA-RISC, and the Intel X86. The two main components of code generation are the instruction scheduler and the register allocator [10]. Several scheduling models exist, including acyclic global scheduling [11],[12], sentinel scheduling [13], and software pipelining using modulo scheduling [14]. In addition, a scheduling technique capable of exploiting architectural support for MCB data speculation exists [2],[3],[15]. The focus of this thesis is to obtain memory profile information for developing a more aggressive MCB data speculative scheduling model.

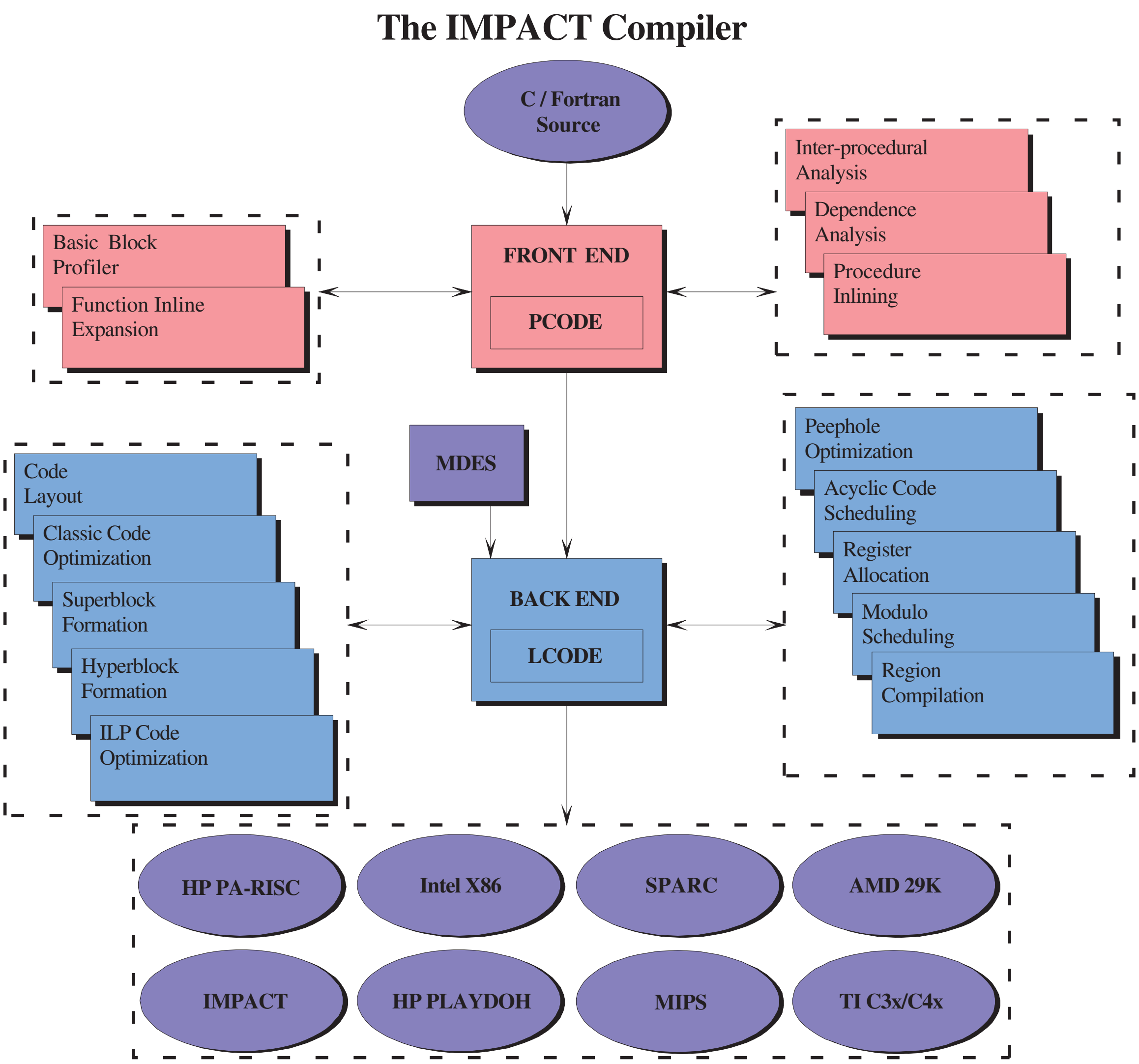

Figure 2.1: The IMPACT Compiler

The IMPACT and HPL Playdoh [16] architectures are two experimental ILP architectures that are also supported. These experimental architectures provide the necessary framework for advanced compiler and architecture research. In fact, the specifications for these architectures are based on parameterizable resources which allow a wide variety of machines to be explored. The IMPACT compiler is able to meet these specifications by using the technology of a machine description database, *Mdes* [17]. The *Mdes* contains a large set of information to assist with optimization, scheduling, register allocation, and code generation. Information such as the number and type of available function units, size and width of register files, instruction latencies, instruction operand constraints, addressing modes, and pipeline constraints, is provided by the *Mdes*. In addition, both architectures support forms of speculative and predicated execution. For this thesis, all experiments utilize the IMPACT architecture with MCB support and various machine formations.

2.2 Overview of the IMPACT Simulation Environment

The IMPACT simulation environment provides the necessary facility to evaluate the effects of compiler techniques on various research architectures. Figure 2.2 presents a block diagram of this environment. The simulation environment involves three important technologies: emulation, probe insertion, and the performance simulator.

Emulation provides a program with the appearance of architectural functions which are not directly implemented by the target machine. For instance, predicated execution

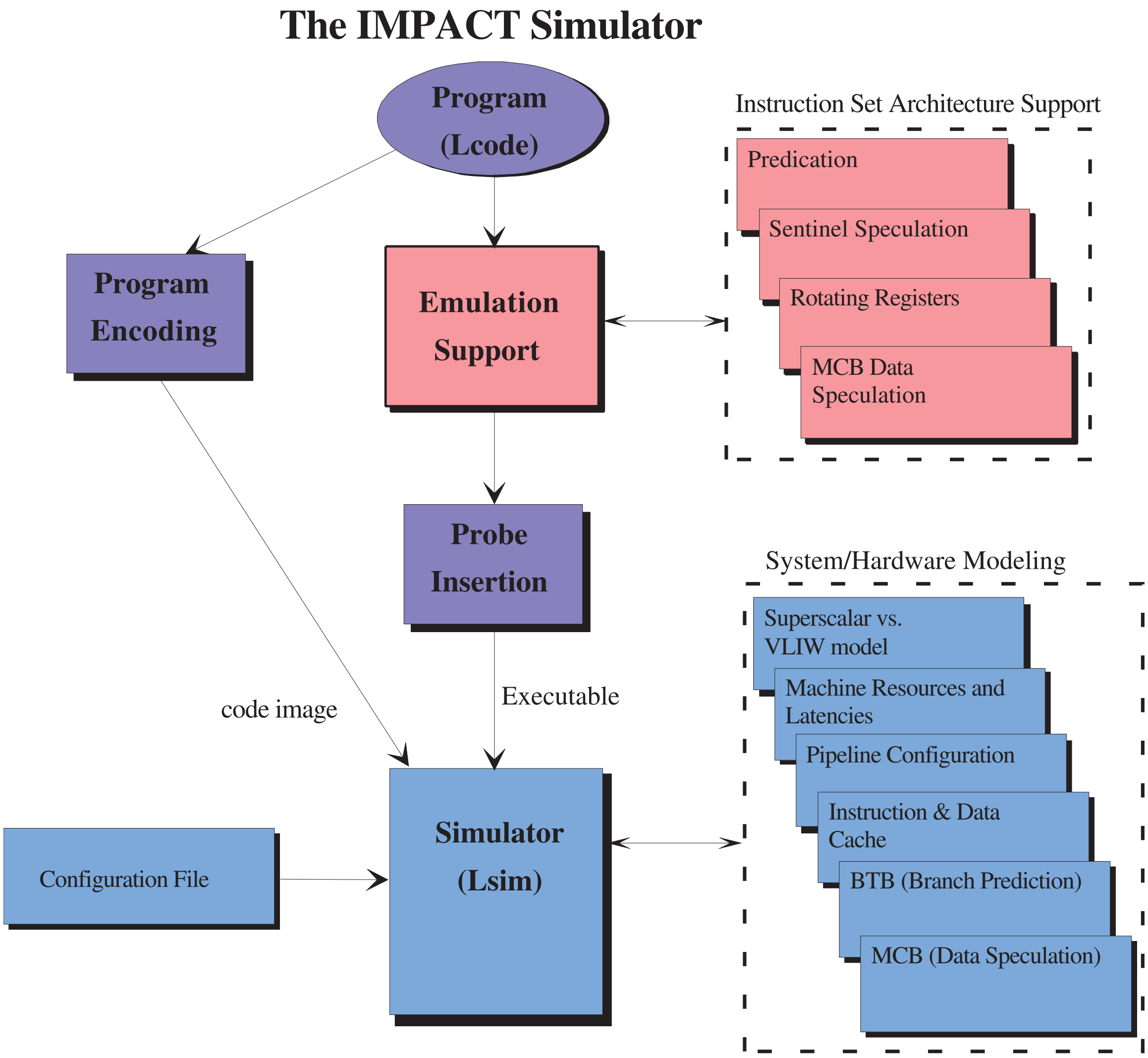

Figure 2.2: The IMPACT Simulation Framework

must be emulated for IMPACT *Lcode* because current processor technology does not include predication support. Other examples of emulated features are speculative execution, VLIW semantics, rotating registers, MCB functionality, or even entire instruction set architectures. Details on IMPACT's MCB emulation support are given in previous work [2].

Probe insertion into program code allows trace information to be generated while the executable is running. Trace information consists of memory target addresses, branch directions, predicate value generations, and jump target addresses. The IMPACT simulator *Lsim* is a trace-driven framework which is capable of cycle-by-cycle simulation of cache memories, branch prediction hardware, instruction pipelines, superscalar and VLIW processors, and MCB hardware. *Lsim* processes trace information and coordinates it with an encoded version of the *Lcode* files to determine executed instructions and accessed memory addresses, in order to properly update the simulated machine structures. For instance, trace information containing branch directions is used by *Lsim* to update simulated BTB structures as well as to follow the proper path of execution. This coordination allows *Lsim* to attribute variable machine cycles to the operations implied by the trace information. In short, the probed executable provides the functionality of the program being simulated, while *Lsim* estimates the target machine cycles. The experimental performance results in this thesis are generated using the IMPACT simulator environment.

2.3 Dependence Analysis

Ambiguous memory dependences can restrict ILP optimizations and scheduling strategies similar to the way that control flow operations deter some transformations. To overcome the problem of ambiguous memory dependences, compilers can perform memory disambiguation, also referred to as dependence analysis. Studies concerning the effect of dependence analysis on performance demonstrate that such practices can result in two to three times performance speedup [2].

Dependence analysis attempts to determine the relationship between memory references at compile time. When successful, the compiler then uses this information to direct code transformations. However, there are many occasions within programs that analysis techniques cannot accurately determine the dependence relationships. Examples include indirect references, non-linear references, or pointer references. Likewise, static analysis of a program is not able to accurately relate dependences to the dynamic control flow of a program, which results in occasional dependences. For example, in Figure 2.3 there are two control flow paths within a section of a program. Each path assigns the address of a variable to the pointer p. These assignments effectively alias p with the accesses of variables x and y. In control block D, the pointer p is deferenced, resulting in an ambiguous dependence with the addition operation. If the control path ABD is taken, then a true memory flow dependence exists through variable x between the addition operation and the deferenced pointer p. However, path ACD would assign the pointer p to variable y, and thus the addition operation would not have a dependence.

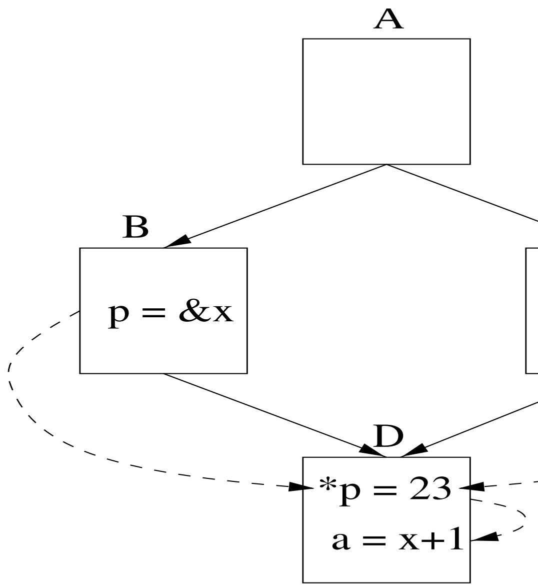

Figure 2.3: Analysis Resulting in Conservative Dependences

In such cases, the results of analysis are conservative, which may cause performance loss in optimizations and scheduling. One appealing aspect of data speculation based on memory dependence profiling is that it can direct optimizations to ignore this conservative analysis. In the same respect, this thesis advocates that dependence analysis can direct memory profiling techniques to the areas of analysis which are known to be inadequate.

2.3.1 The IMPACT Memory Dependence Tracking System

The MDTS (Memory Dependence Tracking System) which the IMPACT compiler uses is commonly referred to as *Sync Arcs* [2]. Dependence analysis is performed once at the *Pcode* level, where source information required for some dependence analysis techniques exists. The result of this analysis is then passed down to the *Lcode* representation.

The term *Sync Arc* derives its name because it maintains the analysis information in the form of synchronizations arcs between *Lcode* memory operations. The scope of this thesis does not permit adequate explanation of *Sync Arcs*. However, it is important to note that *Sync Arcs* possess two characteristics that relate to memory dependence profiling. First, *Sync Arcs* represent all possible memory dependences, and thus are conservative. The second important characteristic of *Sync Arcs* is that *Lcode* modules accurately maintain their information through all compiler transformations. This thesis will illustrate how these characteristics encourage the use of memory dependence profiling within a compiler's infrastructure.

2.4 The MCB Data Speculation Approach

Dynamic memory disambiguation is one potential approach to overcoming aliased memory dependences and increasing ILP for a given schedule. The Memory Conflict Buffer (MCB), proposed in Chen's thesis [15], is one such method. In short, the MCB approach uses hardware buffers to capture run-time conflicts between load and store memory addresses. Additionally, the MCB approach involves the introduction of two new instructions: a *data speculative preload*, which executes as a normal load and additionally signals the hardware that a potential dependence conflict may exist for this load; and a *check*, which scans the hardware buffers to determine if a violation has occurred. Any violations require the *check* to function as a conditional branch, retargeting the program execution stream to the specified correction code. Figure 2.4 illustrates an

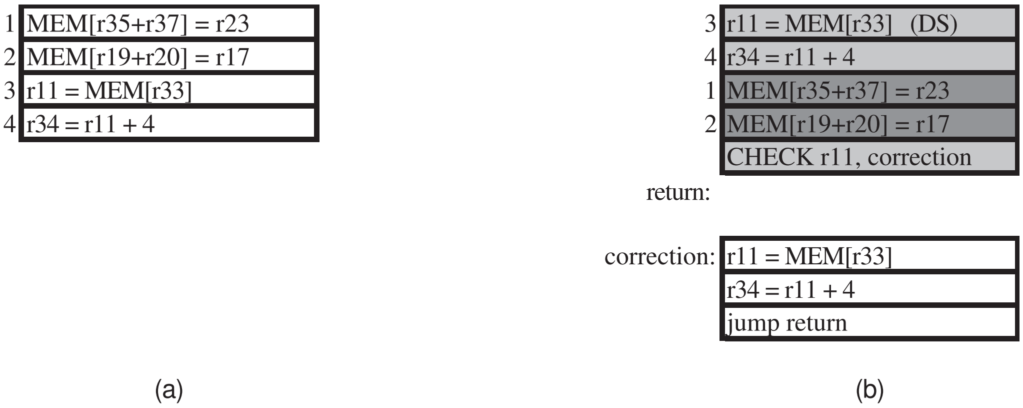

Figure 2.4: Memory Conflict Buffer Example

MCB code example. Upon scheduling the original code 2.4(a), the load operation, *op 3*, has two ambiguous dependences with the store operations *op 1* and *op 2*. These dependences must be obeyed when scheduling the operations for execution, even though a true memory flow dependence may not exist. However, with MCB support, the compiler can perform operation reordering of the load operation and the ambiguous store operations as demonstrated in Figure 2.4(b). Note that the load has been changed to a *data speculative preload*, and that a *check* has been inserted in the original location of the load.

The MCB architectural support of a *data speculative preload* and a *check* allows the compiler to schedule loads above ambiguous stores, thus performing data speculation. This method gives the MCB hardware the responsibility of detecting reference conflicts and invoking correction code. In order to detect conflicts, the MCB hardware records address information for each *data speculative preload*. The memory target of every subsequent store operation is then compared to address information within the MCB to determine whether a conflict has occurred. The hardware records the occurrence of

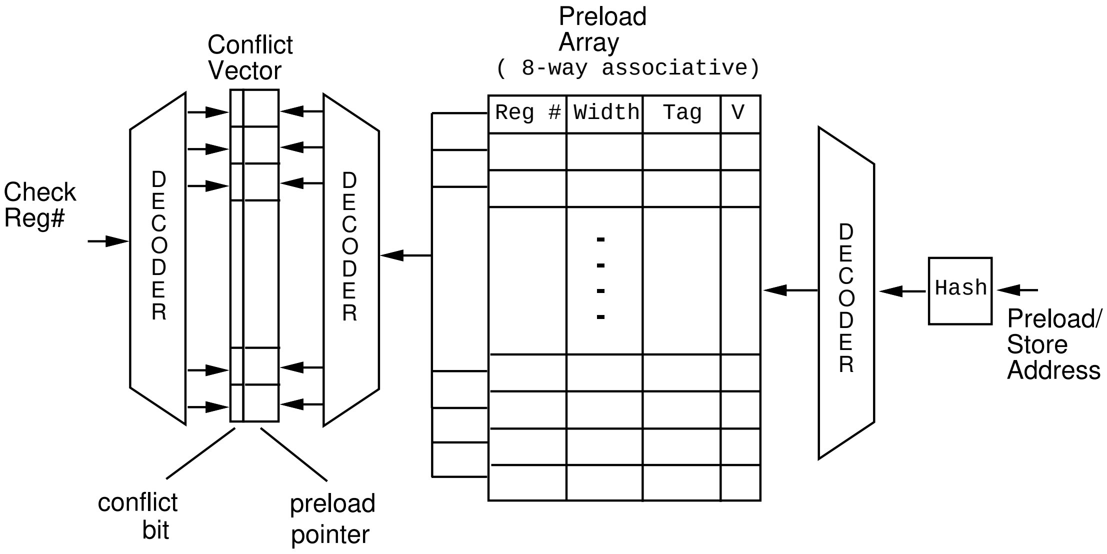

Figure 2.5: MCB Design

conflict. When a conflict entry is accessed by a check instruction, the instruction will execute as a branch to the correction code, the purpose of which is to obey ambiguous store dependences, hence executing the original program order.

The MCB hardware is similar to a small cache, and thus there exist many design alternatives, mostly dealing with address associativity. The simplest of MCB designs is presented in Figure 2.5. The hardware consists of two primary structures, the preload array and the conflict vector, which correspond to the memory address information and the conflict determination, respectively.

The preload array is a set-associative cache structure with entries that contain four fields: (1) the preload destination register number; (2) the preload access width; (3) an address *signature* or *tag*; and 4) a *valid* bit indicating whether the entry currently contains valid data. The preload register field identifies the register number of the preload destination. The access width field indicates the size of memory access of the preload instruction. The address signature is a hashed version of the preload address. Different hashing methods and signature sizes have been explored [2],[3],[15]. The valid bits are reset upon execution of the corresponding check instruction, using the pointer within the conflict vector. Accesses to the preload array are performed using the virtual address of store and preload memory targets.

The conflict vector is equal in length to the number of physical registers, having one entry corresponding to each machine register. Each entry contains two fields, the conflict bit and the preload pointer. The conflict bit is used to record that a conflict to the register of a preload has occurred. The preload pointer specifies which preload array line currently holds the preload associated with this register and allows the preload entries to be invalidated by the check instruction. There are two types of conflicts with the MCB approach: true and false. True conflicts occur as a result of a store target matching the memory address of an advanced load. On the other hand, false conflicts can be the result of insufficient MCB resources that fail to provide unique entries for each data speculative preload operation.

There are numerous design alternatives for lowering the number of false conflicts within an MCB system; however, true conflicts typically result from a compiler's unknowing creation of unprofitable data speculative instances. This thesis will explore using memory profiling to lower the occurrence of true conflicts within schedules that utilize MCB data speculation, thus improving performance.

3. DATA SPECULATIVE OPTIMIZATIONS

There are many opportunities to apply data speculative optimizations in addition to using data speculation during scheduling. Section 3.1 presents the data speculative optimizations of loop invariant code removal and load-subroutine advancement, as related to the preload-verify model. Section 3.2 then describes some of the compiler issues involved with data speculation.

3.1 Preload and Verify

A memory load operation may take several CPU cycles to complete, and is often on the critical path of a body of code. In fact, when a load incurs a cache memory miss, the latency may be even greater. For these reasons it is important to utilize optimizations that hide the latency of load operations. One such optimization is the use of a pair of operations, preload and verify. Previous work [15] has already investigated preload operations and cache performance; however, benefits exist in using preload and verify operations in data speculation instances.

`ST (a1),r1`
`r1 = LD(a2)`
	`r2 = LDS(a2)`
	`ST (a1),r1`
	`r2 = LDV(a2)`

(a)	(b)

Figure 3.1: Preload-Verify Operations: (a) Original Code, (b) After Use

The HPL Playdoh [16] architecture provides runtime memory disambiguation support with three operations: the data speculative load (LDS), the data verify load (LDV), and the data verify branch (BRDV). The Playdoh LDS is similar to the MCB preload, and the Playdoh BRDV is analogous to the MCB check. However, the LDV operation behaves like a conditional load operation predicated on whether a data speculative conflict has occurred, similar to the preload-update operation in previous work [18].

The LDS-LDV pair is an efficient mechanism for supporting data speculative instances in which dependent operations of the data speculative load are not scheduled before the verification operation. This use differentiates itself from the LDS-BRDV pair and the traditional MCB approach because of its lack of correction code. All that is necessary for correction is that the LDV behave as a load operation. When no correction is necessary, the LDV does not need to perform any function.

Figure 3.1 illustrates the use of the preload-verify pair. The primary advantage is that the LDS operation is now scheduled before the store operation (ST) and the load's subsequent dependent operations are scheduled to get their source operands from the LDV operation. As such, the compiler schedules these dependent operations at the cycle in which the LDV verifies the data speculative load. In most models, this verification requires a single cycle, which is much less than normal load operation latencies. When no conflict occurs, the overall effect is that the critical path has been reduced. However, when verification signals a conflict, the LDV functions differently. First the processor pipeline is flushed and then the LDV is reissued as an actual load operation. The subsequent dependent operations must wait until this load operation completes before executing. The performance degradation due to conflicts illustrates the need for an accurate mechanism for directing the use of data speculation, which is the subject of this thesis.

The following two subsections will explore using the preload-verify mechanism as the basis for creating more advanced data speculative optimizations.

3.1.1 Loop invariant data speculative optimizations

The goal of the traditional loop invariant code removal optimization is to move operations which compute invariant results from the loop body to the loop header. It is often the case that a load operation within a loop is invariant. However, optimizing compilers may not know that such loads are invariant due to ambiguous store operations present within the loop. Consequently, the efficiency of the loop invariant optimization depends partially on the accuracy of a compiler's memory disambiguation scheme or its MDTS.

For loop instances which contain potentially invariant loads, the preload-verify data speculation approach can be applied to facilitate the removal of such load operations from the loop body. Figure 3.2(a) illustrates a general loop with this behavior. The basic meaning of a load operation requires that, at that point in the loop, the value held

```
                                      r2 = LDS(a2)
for(i = 0; i < 100; i = i + 1)        for(i = 0; i < 100; i = i + 1)
{                                     {
     ST (a1),r1                           ST (a1),r1
     r2 = LD(a2)                          r2 = LDV(a2)
}                                     }

        (a)                                   (b)
```

Figure 3.2: Data Speculative Loop Invariant Optimization (a) Original Code, (b) After Optimization

by the load destination register must contain the data at the load address in memory. Therefore, the load can be safely removed as long as the meaning of the load is preserved for that point in the loop.

Figure 3.2(b) shows the load operation advanced to the header of the loop, while a verify operation has been placed at the load's original position. The verify operation is used to check whether the load value has changed as a result of the ambiguous stores.

In order for the data speculative loop invariant optimization to be applied for any general loop, several conditions must first be satisfied by the potentially invariant load.

- *Condition 1:* The load's address must be loop invariant in the traditional sense.
- *Condition 2:* The loaded value must only be used in blocks dominated by the load.
- *Condition 3:* The load must be the only load within the loop to define the destination register.

Together, these conditions ensure that this data speculative optimization can be applied without changing the original behavior of the loop. Condition 1 is necessary for the

MCB hardware to track a data speculated load address with store operations throughout the loop execution. Condition 2 must be satisfied so the preload operation does not disrupt the original use-def dataflow chain of the program. For instance, when the destination register of a preload is already holding a value to be used within the loop, further code transformations are required for the data speculative optimization to work properly. Thus, Condition 2 is a conservative condition that selects only opportunities within loops that do not require further transformations. Lastly, Condition 3 is necessary so that the MCB conflict vector is not incorrectly cleared by other load operations within the loop.

The use of the data speculative verify operation within a loop requires additional support than when speculatively executing within acyclic code. For the case when the verify does not determine that a conflict occurred, its corresponding MCB entry must remain active to perform verifications for future iterations of the loop. Upon recognizing a conflict, cyclic verifications need to perform the necessary reloads and also renew the validity of their MCB entry. This functionality can easily be supported as a separate data speculative operation, referred to as a *verify loop load* [15].

It is important to note that the data speculative loop invariant optimization can be applied more aggressively by allowing the operations dependent on the data speculative load to also advance to the preheader of the loop. This action requires the use of an MCB check operation at the verification point, and the generation of a correction control block. Such an optimization has the potential for greater performance since more operations are removed from the body of the loop. However, the correction process for conflicts is clearly more involved. The overall effect is that the optimization's profitability has a greater dependence on the conflict rate of the data speculative load.

Figure 3.3 describes an algorithm for applying simple, data speculative loop invariant code removal. There are two components of the algorithm, one providing the three conditions described earlier and one analyzing the potential benefits and penalties of applying the optimization. The second component of the algorithm assumes accurate knowledge of the outcome of data speculative verifications. In reality these outcomes cannot be determined until run time. Nevertheless, the algorithm illustrates the difficulty of using data speculation prosperously, and helps demonstrate the usefulness of memory profiling presented in the next chapter. Examples of benefit and penalty estimation functions for data speculation within loops are:

$$\begin{aligned} Benefit &= E \cdot (1 - C) \cdot Save \\ Penalty &= E \cdot C \cdot Cost \end{aligned}$$

Both functions include estimates of the loop execution frequency E and the conflict rate C. In this instance, the conflict rate is defined as the percentage of time that the load verify must reload data due to an ambiguous store operation conflicting with the presumed invariant load.

In addition, the functions employ approximations for the height reduction benefit $Save$ when no conflict occurs, and the cycle loss $Cost$ attributed to flushing the pipeline

```
data_speculative_loop_invariant_code_removal(function)
{
        candidate_ops = NULL
        for each loop in function, loop_i {
                for each control block in loop_i, cb_j {
                        for each op in cb_j, op_k {
                                /* Functionality Conditions
                                if !Loop invariant operands (op_k) then
                                        continue
                                if !Unique definition in loop (op_k) then
                                        continue
                                if !Def reaches all operations out of loop (op_k) then
                                        continue

                                /* Benefit and Penalty Comparison */
                                if (Advancement benefit < Call Conflict penalty)
                                        continue
                                if (Advancement benefit < Store Conflict penalty)
                                        continue

                                        candidate_ops = candidate_ops + op_k
                        }
                }
        }
        /* Optimal ordering of candidates by benefit */
        sort candidate_ops by benefit

        num_advanced = 0
        for each op in candidate_ops, op_i {
                if (num_advanced > MAX_ADVANCEMENT) then
                        break
                if (maximum_possible_loop_PDC(op_i) > MAX_LOOP_PDC) then
                        break
                Copy op_i to new_op_i in loop preheader
                Change new_op_i to PRELOAD
                Change op_i to LOAD_VERIFY
                num_advanced = num_advanced + 1
        }
}
```

Figure 3.3: Algorithm for Data Speculative Loop Invariant Removal

when a conflict occurs. The value of $Cost$ can be adjusted to reflect the target machine's handling of conflicts. In the same manner, the $Save$ value is an estimation of the savings achieved by scheduling LDV operations, which have a smaller latency when no conflict occurs, in place of normal LD operations.

The second half of the loop invariant algorithm is used to consider all individual conflict information as a single conflict. Individual profile information for the loop invariant optimization can be misleading. For example, profile information could indicate that a load operation within a loop has conflicts with three stores, with respective conflict rates of 5%, 10%, and 15%. However, it cannot always be determined from the control flow of the loop whether these conflicts occur during the same iteration. Therefore, from this information, two estimates can be made for the overall loop conflict rate:

$$
\begin{aligned}
OptimisticEstimate &= \max(PDC_1, \cdots, PDC_n) \\
&= \max(5, 10, 15) = 15 \\
PessimisticEstimate &= \sum_{i=1}^{n} PDC_i \\
&= \sum(5, 10, 15) = 30
\end{aligned}
$$

The optimistic estimate corresponds to the case in which the conflicts of the three ambiguous stores overlap. For instance, all three store operations may invalidate the load during the same iteration. In this case, only one conflict occurs, thus the overall conflict rate estimate is the maximum of the individual conflict rates. Accordingly, the pessimistic

```
buffer_ptr = buffer[ci][yindex+yoffset] + start_col;
for (xindex = 0; xindex < compptr->MCU_width; xindex++) {
  coef->MCU_buffer[blkn++] = buffer_ptr++;
}
```

Figure 3.4: Loop Invariant Code Removal Optimization Example in *compress_output* of *132.ijpeg*

estimate corresponds to the case in which the conflicts of the three store operations are exclusive, thus a conflict occurs for each individual conflict. In this case, the conflict rate estimate is the sum of all individual conflict rates. This example illustrates that profile information must be used with care in compiler optimizations.

There exist many opportunities for using the loop invariant code removal optimization within the SPEC92 and SPEC95 benchmark suites. For example, Figure 3.4 shows a loop within the function *compress_output* of the benchmark *132.ijpeg* that contains loop invariant code. In this case, the load of *compptr->MCU_width* for the loop exit condition must be performed for each loop iteration because of an ambiguous store to *coef->MCU_buffer[blkn++]*. It is important to note that with aggressive pointer analysis, the load would be determined as loop invariant. Without such analysis, the use of data speculation could be used to treat the load as loop invariant, thereby generating more efficient code. Overall, the example of Figure 3.4 illustrates the ability of data speculation to provide performance improvements in commonly occurring loops.

```
                          r1 = LDS(a1)
CALL function()           CALL function()
r1 = LD(a1)               r1 = LDV(a1)

    (a)                       (b)
```

Figure 3.5: Load-Subroutine Advancement Optimization (a) Original Code, (b) After Optimization

3.1.2 Load-subroutine advancement optimization

In addition to store operations, subroutine calls can cause ambiguous memory dependences which limit ILP. Some of these dependences can be removed by performing interprocedural analysis [2]. Consequently, the implementation of conservative analysis techniques will result in the loss of potential performance. Additionally, in many compilation environments, subroutine code may not be available for global program analysis. In such cases there exists an opportunity to speculatively execute a load before a potentially aliased subroutine call. The load-subroutine advancement data speculative optimization is illustrated in Figure 3.5.

There are two major components in the algorithm of Figure 3.6 for effectively utilizing the load-subroutine advancement optimizations. First, similar to all data speculative optimizations, it is important to initiate the optimization when the probability of conflict is low. This involves weighing benefits and penalties in a similar manner to the previously discussed algorithms. Unlike traditional optimizations which attempt to remove unnecessary operations, ILP and speculative optimizations can often increase the operation count and register lifetimes in a region of code. The advancement of a load

```
data_speculative_load_subroutine_advancement(function)
{
    for each loop in function, loop_i {
        for each control block in loop_i, cb_j {
            for each op in cb_j with subroutine control dependence, op_k {
                if Data flow from subroutine to (op_k) then
                    continue
                if (Advancement benefit < Call Conflict penalty)
                    continue
                if (Advancing op_k increases register pressure)
                    continue

                /* Apply Data Speculation */
                Copy op_k to new_op_k above subroutine
                Change new_op_k to PRELOAD
                Change op_k to LOAD_VERIFY
            }
        }
    }
}
```

Figure 3.6: Algorithm for Data Speculative Load Subroutine Advancement

operation above a subroutine call increases register lifetimes and could result in register spilling, thus lowering performance. As such, the algorithm in Figure 3.6 attempts also to weigh the benefit of this data speculative optimization by using an estimation of the added register pressure.

Several opportunities for using the load subroutine advancement optimization exist within the SPEC95 benchmark suite. For example, Figure 3.7 shows a section of frequently executed source code from function *lupdate* of the benchmark *099.go*, which illustrates a load operation being executed below a subroutine call. The subroutine

```
++numnodes;
c = mvcolor[upptr];
l = 0;
m = 0;
nlist = EOL;
i = fdir[s];
for(ldtmp = ldir[i]; i < ldtmp; ++i){
  sn = s + nbr[i];
  ++lnbf[sn][c];
  --lnbn[sn];
  dellist(s,&nblbp[sn]);    /* SUBROUTINE */
  g = board[sn];            /* LOAD-ADVANCEMENT */
  if(g == NOGROUP){
     lnew[m++] = sn;
  }
 }
```

Figure 3.7: Load Subroutine Advancement Optimization Example in *lupdate* of *099.go*

call *dellist* causes a false memory dependence with the access of the global array *board*. Without interprocedural analysis, the compiler must assume that any element of the array *board* could be defined in *dellist*. In this case, data speculation support would allow the load to be executed earlier, thus improving performance since the array is not changed within the *dellist* function.

3.2 Compiler Issues

The IMPACT compiler's scheduling techniques are currently capable of exploiting architectural support for MCB data speculation [3]. However, the methodology does not support the advanced data speculative optimizations described in Sections 3.1.1 and

3.1.2. Furthermore, the existing MCB scheduling methodology has several drawbacks which are easily observed from the algorithm shown in Figure 3.8.

The first drawback within the algorithm is that all ambiguous memory dependences are removed and MCB check operations are applied. This illustrates the simplicity of the current scheduling methodology, because there is no mechanism which evaluates individual dependences. Without question, only the ambiguous memory dependences with low probabilities of conflict should be removed when performing optimal MCB scheduling. Another inherit problem with the current methodology is that it schedules two versions of each selected control block and then selects the schedule with the best performance estimate. This action drastically increases compilation time. Finally, the algorithm has only the ability to use *check* operations, not LDV operations. In the scheduling instance when only a load is advanced above an ambiguous store operation, the use of a *check* instead of an LDV operation to perform conflict detection can degrade performance.

This thesis advocates replacing the current scheduling methodology with an algorithm that removes dependences using a profile-based conflict probability. The new algorithm is illustrated in Figure 3.9 and will be evaluated in Chapter 5 of this thesis. The algorithm has the ability to initiate data speculation in the presence of dependences with low conflict rates. In this thesis this rate is set to 15% as a general starting point to allow an aggressive, optimistic use of data speculation support. Future work will experiment

```
schedule_cb(cb)
{
        initialize_priority_ready_queue(cb)
        while(ops in cb are unscheduled) {
                for each in priority_queue, op_i {
                        if (can_schedule_op(op_i) {
                                schedule_op(op_i)
                                remove_op_dependences(op_i)

                                if (op_i is DATA_SPECULATIVE_LOAD) and
                                        (op_i is scheduled below all ops in store_list of op_i) then {
                                                remove_check_op_from_scheduling_queues_for_load(op_i)
                                                remove_check_op_dependences(op_i)
                                }
                        }
                }
                /* Move candidate ops from ready_queue to priority_queue */
                update_priority_queue()
                /* Move candidate ops from not_ready_queue to ready_queue */
                update_ready_queue()
        }
}
old_MCB_scheduling_algorithm(function)
{
        for each loop in function, loop_i
                for each control block in loop_i, cb_j
                        build_operation_dependences(cb_j)
                        scheduled_cb_j = schedule_cb(cb_j)
                        if profile_execution_weight(cb_j) > MCB_MIN_WEIGHT then {
                                mcb_cb = cb_j
                                remove_all_ambiguous_dependences(mcb)
                                maintain_store_list_for_each_load(mcb)
                                add_mcb_code(mcb)
                                scheduled_mcb_cb = schedule_cb(mcb)
                                perf_mcb = estimate_performance(scheduled_mcb_cb)
                                perf_cb = estimate_performance(scheduled_cb)
                                if (perf_mcb - perf_cb) > MCB_MIN_SPEEDUP then
                                        scheduled_cb_j = scheduled_mcb_cb
                        }
}
```

Figure 3.8: Existing Algorithm for MCB Scheduling

```
new_MCB_scheduling_algorithm(function)
{
        for each loop in function, loop_i {
                for each control block in loop_i, cb_j {
                        if profile_execution_weight(cb_j) > MCB_MIN_WEIGHT then {
                                build_operation_dependences(cb_j)
                                remove_dependences_with_low_conflict_rates(cb)
                                maintain_store_list_for_each_load(cb)
                                add_mcb_code(cb)
                                scheduled_cb_j = schedule_cb(cb)
                        }
                        else {
                                build_operation_dependences(cb_j)
                                scheduled_cb_j = schedule_cb(cb_j)
                        }
                }
        }
}
```

Figure 3.9: Algorithm for Scheduling

```
r1 = LD(a1)
r3 = LDS(a1)
ST (a2),r2
r3 = LDV(a1)

    (a)
```

```
r1 = LD(a1)
ST (a2),r2
r3 = LDV(a1)

    (b)
```

Figure 3.10: Redundant Load Elimination after Data Speculation: (a) Original Code, (b) After Optimization

with the value of this rate to investigate the construction of optimal data speculative schedules.

3.2.1 Compiler difficulties

The initiation of data speculative optimizations complicates several areas of a compiler's infrastructure. First, data speculative operations affect the concepts of dataflow and control flow, two fundamental parts of compiler technology. Likewise even the simplest transformations within a compiler must maintain the properties of data speculative operations. For instance, Figure 3.10 demonstrates how the traditional, redundant load elimination optimization may unknowingly destroy the meaning of a data speculative optimization. In this case, a preload operation and a load operation are transformed through redundant load elimination into a single load operation. However, this destroys the data speculative optimization. Instead, the elimination optimization should transform the pair of operations into a preload operation.

Another complication of data speculative optimizations is the generation of correction control blocks. The formation of any new control block changes the original control flow of a program. Likewise, new dataflow analysis must be performed to include these created blocks. In addition, correction blocks give the compiler the added responsibility of tracking the data speculative dependent operations of a data speculative load through all compilation phases.

Data speculation also disrupts the traditional dependence graph generation for operations within a control block. Data speculative loads differ from their original load operations in that certain memory dependences have been removed. These dependences are effectively transferred to the verify operations of the data speculative load. Likewise, the control dependences of the original load must also be transferred. Traditional scheduling algorithms generate all dependences before scheduling. However both data speculative scheduling algorithms described in Section 3.2 insert and remove data speculative operations and their dependences during scheduling. Hence, such algorithms present difficulties in maintaining valid dependences between operations. For all of these reasons, data speculation is viewed as a set of complex transformations which require an advanced compiler infrastructure. Due to the limited scope of this thesis, innovations on the infrastructure of such transformations are left as the topic of further work.

4. MEMORY DEPENDENCE PROFILING

As explained in the previous chapter, data speculation is a powerful technique for exploring the full potential of ILP processors by speculatively executing a load before potentially conflicting stores. However, frequent conflicts between advanced load and store operations will result in a performance loss attributed to the overhead of executing correction code. Memory dependence profiling is one approach for finding profitable data speculation opportunities within a program.

The objective of memory dependence profiling can be accomplished in many ways with differing accuracy and difficulty. This chapter identifies three classes of memory reference profiling: address-based, index-based, and collection using program instrumentation. The aspects which distinguish the profiling classes are profiling time, accuracy, and the scope of the profiling information. This chapter begins by illustrating the relationship between the potential data conflict rate and data speculation. Then the three classes of profiling are described. Finally, the actual implementation of the profile technique of this thesis is explained.

4.1 Profile Data Conflict (PDC) Rate and Data Speculation

The Profile Data Conflict (PDC) rate is defined here to be the number of times for a given profile that a store operation conflicts with a load operation relative to the number of times the load operation executes. The term "conflicted" describes an instance in which a particular store operation provides the result for a subsequent load operation. The algorithms presented in Section 3.2 utilized benefit estimation functions for initiating data speculation based on estimates of load-store conflict rates. Although compilers can effectively approximate the cost and savings within these benefit functions, accurate estimation of conflict rate is difficult.

One possible benefit function [19] for estimating the profitability of data speculation is derived below, where CC is the correction cost overhead, and DS is the data speculative benefit assuming zero conflicts.

$$DSProfit = DS - CC \cdot PDC \tag{4.1}$$

$$DS - CC \cdot PDC \geq 0 \tag{4.2}$$

$$PDC \leq \frac{DS}{CC} \tag{4.3}$$

In Figure 4.1, this profit function is graphed with respect to values of the PDC rate and the $\frac{CC}{DS}$ ratio. The curve establishes two distinct regions representing profitable and unprofitable data speculative execution. The profit curve itself identifies the PDC rate

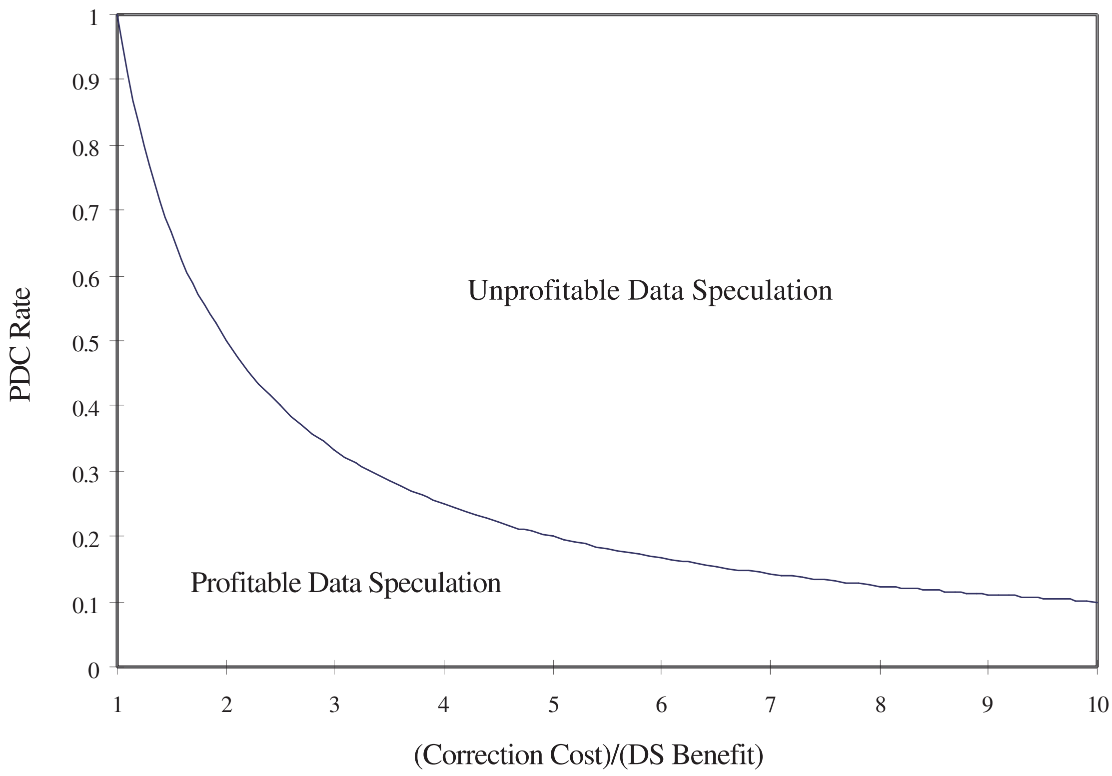

Figure 4.1: Profitability Curve Using the PDC Rate

for a particular $\frac{CC}{DS}$ setting that results in zero profit. The general relation of PDC and $\frac{CC}{DS}$ is that high PDC rates require low $\frac{CC}{DS}$ ratios for beneficial data speculation. The inverse relation is also true. For example, when a PDC rate is less than 10%, the corresponding cost-to-benefit ratio ($\frac{CC}{DS}$) of the data speculative instance could be as high as 10 without penalty.

As mentioned, an advanced compiler may be able to estimate the values of CC and DS, however it is the estimate of the PDC which cannot easily be predicted. Nevertheless, by performing memory profiling, a compiler can establish PDC rates that estimate

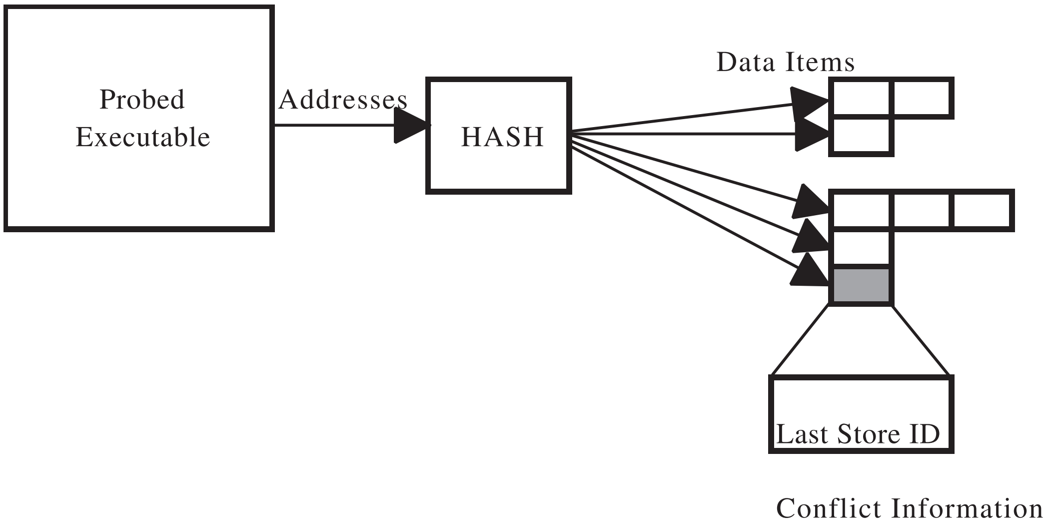

Figure 4.2: The Address-Based Memory Reference Collection Approach

the conflict rates for data speculation. Thus, benefit functions similar to Equation 4.3 can be included within a compiler's data speculative algorithms. The overall result is better selection of data speculative instances within program code, as will be illustrated in the next chapter. The remaining sections of this chapter describe different methods for collecting the PDC rate information.

4.2 Address-Based Conflict Rate Profiling

Conceptually, the address-based memory reference profiling approach is based on the relatively simple hash table principle. The memory addresses of load and store operations in a program's probed executable are extracted and used as the hash keys for a single global hash table. Figure 4.2 represents an overall view of this approach.

The addresses of the hash table entries correspond to specific data items, such as variables or array elements, in the executing program. Since the goal of memory profiling is to associate conflicts between load and store operations, it is necessary for each hash entry to maintain the unique id for the last store operation that modified the respective program address. When load operations execute, their memory addresses access the hash table and find the respective hash entry. This entry contains the store id of the last conflicting store operation, and is used to construct a conflict between the load and store operations. Conflicts between operations are maintained in a separate structure and are updated for each occurring conflict. Previous work [20] has developed a similar address hash structure for evaluating the locality of memory accesses.

Many research facilities only have access to program traces and do not possess the capability to instrument-program code. Therefore, the address-based collection approach provides an effective way to gather memory conflict information in any environment. However, there are several disadvantages to this approach of profiling. The most important disadvantage is that the approach requires an excessive amount of memory for catching all conflicts. Likewise, the process of profiling is extremely time consuming. For these reasons, memory reference profiling environments based solely on the address-based approach are rarely constructed.

4.3 Index-Based Conflict Rate Profiling

The address-based collection method discussed previously performs conflict detection uniformly on all memory operations. Such handling does not have the ability to focus its attention on particular operation pairs, and this causes the excessive profile time. The most important aspect of the index-based profiling approach is the facility to use a list of program operations to guide the profile collection, thus reducing profile time.

The premise for this method is that for each load operation there typically exists a set of interesting stores which cause ambiguous memory dependences. It is often the scheduler which exposes such ambiguous references as problematic; however, any optimization within a compiler is capable of generating lists of operation pairs with ambiguous dependences that deter performance. In fact, a user investigating the performance of a program benchmark may also determine a set of load-store operation pairs for which conflict information would help recognize inefficiencies or bottlenecks. At the same time, some dependences between load and store operations are known to exist statically, and there is no reason for memory profiling to examine such references for conflicts. The index-based profile method can be directed to efficiently ignore these conflicts.

Figure 4.3 illustrates the index-based memory profiling approach. This approach allocates a single conflict entry for each load-store pair selected prior to profiling. Hence, a global hash table structure does not exist. Instead, each memory operation has a list of the conflict entries which must be updated by its execution. For instance, in Figure 4.3, the store operation *ST1* must update the conflict entries for loads *LD1* and *LD2*. Similarly, store *ST2* only updates the conflict entry for load *LD1*. Each conflict entry consists of two fields, the memory address of the store operation and a conflict count. When a load operation executes, the memory address of the store is compared to the load address, and matches result in incrementing the conflict count field.

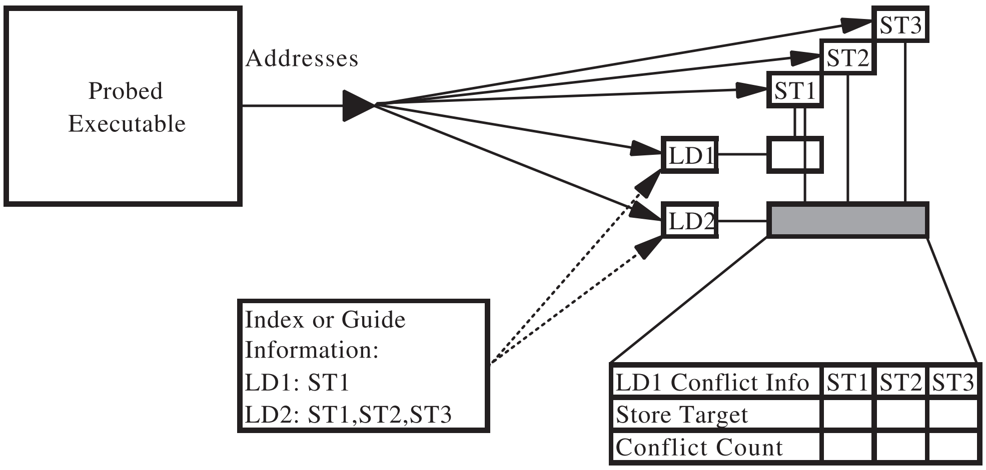

Figure 4.3: The Index-Based Memory Reference Collection Approach

The obvious advantage of this approach is that it offers more directed collection of ambiguous references, ignoring memory conflict information that will not be utilized in optimizations. As such, the method allows users to define which loads are interesting in terms of the compiler's scheduling and optimization scope.

However, the index-based approach has a definite disadvantage. Memory conflict detection takes place on two planes, time and space. Figure 4.4 illustrates an example of the temporal disadvantage of the index-based memory reference collection approach. In this case, there exists one load operation (LD) and one store operation (ST) executing under different conditions within a loop. If the LD and ST cannot be disambiguated, then the ST prevents the LD operation from being loop invariant. In an environment where a compiler was closely integrated with a profiler, the compiler would elect to profile the LD-ST pair in order to provide PDC information for the loop invariant data speculative optimization. Using the index-based approach causes the ST operation to write its target address to the conflict entry of the LD operation. However, several iterations of the loop may execute the ST operation several times before the first LD operation. In this case, the index-based approach only saves the last target address of the store. Thus, the load operation would not be able to collect conflicts that occurred with the store operation several iterations earlier. The overall result is lower PDC accuracy compared to the address-based method. Although the index-based approach could be constructed to save a fixed amount of the last store addresses, the additional profile accuracy gained would come with the cost of increased profile time. This problem is eliminated with the region-based approach that will be discussed in Section 4.4.

```
for (i=0; i < 100; i++) {

   if (condition)
      ST (a1),r1
   else
     r2 = LD(a2)

}
```

Figure 4.4: Index-Based Approach Example

Although the loss of conflict information between iterations is a disadvantage, the index-based approach can be used for collecting profile information within the operation scheduling scope. In the scheduling scope, the conflict collection between operations executing within the same control block are not affected by the temporal conflict collection problem. Therefore, the primary advantage of the index-based approach is accurate and efficient PDC collection within the operation scheduling scope. However, in the case of scheduling a loop block that has been unrolled, the temporal problem of profile information still exists. Thus, the use of index-based profile method requires certain transformations in a compiler to be performed before profiling.

4.4 Program Instrumentation

In this approach, the user defines regions within program code that contain important areas for data speculation. The regions represent the different sections of code which may define new values for load operations and thus can be used to generate PDC rate information. The regions consist of three components, a start point, a set of load invalidate operations, and a load operation or an ending point. The start point of a region indicates the location within a program where a load operation may potentially be relocated. The start point is instrumented by the function call CONFLICT_REGION_START, which is responsible for performing a preload, saving the loaded data, and initializing the region conflict count to zero. The set of load invalidate operations consists of any operation that modifies memory. Finally, a region is ended with the load operation and is instrumented by the function call CONFLICT_REGION_END. This function performs the load operation, and compares the loaded data to the pre-loaded data. The region conflict count is incremented when the two data values are different.

The profile regions can directly correspond to potential uses of MCB data speculation. For instance, the start of a profile region may be the point within code that a compiler would insert preload operations. Other work [15] has advocated a similar approach that profiles MCB uses, but does not use the region definition. However, the region definition improves the usefulness of performing memory profiling and is best observed with an example.

Figure 4.5 illustrates one use of program instrumentation. In this example, the LD operation is ordered below several ST operations. If the compiler cannot statically disambiguate the LD with the ST operations, memory dependence profiling may be able to direct data speculative scheduling techniques. Figures 4.5(b)-(c) illustrate two possible region definitions for the LD operation. Figure 4.5(b) shows a single region and Figure 4.5(c) demonstrates multiple overlapping regions within the original code.

The single region definition will basically return the conflict rate for an instance of data speculation where the preload operation is placed at the region start location CONFLICT_REGION1_START. The conflict rate obtained at that point will involve all four store operations. Although this information is accurate for this particular instance of data speculation, it does not provide precise conflict information between the load and any single store operation.

By defining multiple regions for a load operation, more detailed profile information can be obtained, which is the benefit of using this region definition. Clearly, the number of possible regions for some load operations can be large, and the use of all possible regions

```
ST (a1),r1
ST (a2),r2
ST (a3),r3
ST (a4),r4
r5 = LD(a5)

    (a)
```

```
CONFLICT_REGION1_START(a5)
ST (a1),r1
ST (a2),r2
ST (a3),r3
ST (a4),r4
CONFLICT_REGION1_END(a5)
r5 = LD(a5)

   (b)
```

```
CONFLICT_REGION1_START(a5)
ST (a1),r1
ST (a2),r2
CONFLICT_REGION2_START(a5)
ST (a3),r3
ST (a4),r4
CONFLICT_REGION1_END(a5)
CONFLICT_REGION2_END(a5)
r5 = LD(a5)

      (c)
```

Figure 4.5: The Use of Conflict Regions (a) Original Code, (b) One Conflict Region, (c) Multiple Conflict Regions

would increase the overall profiling time. Obviously, when profiling to determine loop invariance there is little loss of profiling precision if a store operation that rarely produces a conflict is included in a region with several store operations that consistently produce conflicts. However, when profiling for the scheduling scope, this is not the case. This raises the issue of whether an optimal set of regions exist for collecting PDC information for a particular load, which is the focus of previous work [19]. Clearly there is a tradeoff between collection time and the precision of the collected information. Likewise, the the precision of information depends on whether the profile system is being used to guide data speculative algorithms or to judge the benefit of an already initiated data speculative code.

Region identification and profile collection are not limited to the scheduling scope. Figure 4.6 illustrates examples of region definitions for the data speculative optimizations of loop invariant code removal and load-subroutine advancement. Figure 4.6(b) represents another approach that uses the CONFLICT_REGION_END to simply track whether a conflict occurred and not keep track of the conflict count. This approach would be able to place the region end marker outside the loop, resulting in more efficient profiling. However the profile information would not designate the number of conflicts as could the approach in Figure 4.6(a). The difference between the approaches is related to the function supported by instrumentation of the region end.

For several reasons, the region information is more helpful for optimization algorithms than scheduling algorithms. For instance, the program instrumentation approach is an effective mechanism for retrieving interprocedural PDC information because it does not process memory addresses to determine conflicts. Trace-based profile methods process the list of addresses and assign conflicts between pairs of operations. However, in the load-subroutine optimization, all that matters is whether the subroutine causes conflicts; more specific information is not necessary. Another advantage of the region profiling approach is that the problem with individual profile information and the loop invariant optimization presented in Chapter 3 does not occur. It is important to note that the CONFLICT_REGION1_END instrumentation in Figure 4.6(a) also behaves as a CONFLICT_REGION1_START, since it is included inside a loop. This is necessary for the instrumentation to determine conflicts throughout all iterations of the loop.

```
CONFLICT_REGION1_START(a2)
for(i = 0; i < 100; i++ )
{
   ST (a1),r1
   CONFLICT_REGION1_END(a2)
   r2 = LD(a2)
}
```

(a)

```
CONFLICT_REGION1_START(a2)
for(i = 0; i < 100; i++ )
{
   ST (a1),r1
   r2 = LD(a2)
}
CONFLICT_REGION1_END(a2)
```

(b)

```
CONFLICT_REGION1_START(a1)
CALL function()
CONFLICT_REGION1_END(a1)
r1 = LD(a1)
```

(c)

Figure 4.6: Examples of Regions for Optimizations (a) Loop Invariant, (b) Load-Subroutine Advancement

The primary advantage of this approach is that the profiling time is much shorter compared to the trace driven approaches. However, there is one obvious disadvantage: the technology is more complex because it involves altering the original program code and determining regions.

4.5 *Sync Arc* Companion Profiling

Each of the memory reference profiling approaches described previously provides a different amount of accuracy, based on the characteristics of its design. Table 4.1 lists the primary advantage of each profiling approach.

Table 4.1: Summary of Advantages of Memory Collection Approaches

Profiling Method	Primary Advantage
Address	Accurate Collection
Index	Efficient Collection
Region Oriented	Effective Profiling for Data Speculative Optimizations

This thesis develops a combined memory reference infrastructure from the investigation of each of the profiling approaches previously presented. The purpose of the infrastructure is to gather memory reference information for improving all data speculative algorithms within the IMPACT compiler. Several elements are necessary to achieve this goal. First, the profiling information must be maintained throughout all compiler transformations. This is an important requirement for the infrastructure, because it is desirable to have correct profile information through several phases of a compiler, without needing to repeat the profiling process. Another requirement is that the user must be able to confine the profile collection to interesting operations. This is particularly useful, because a compiler could then only profile the ambiguous memory dependences.

All of these requirements contributed to the construction of the profile system illustrated in Figure 4.7, called *Sync Arc Companion Profiling.* This approach uses the *Sync*

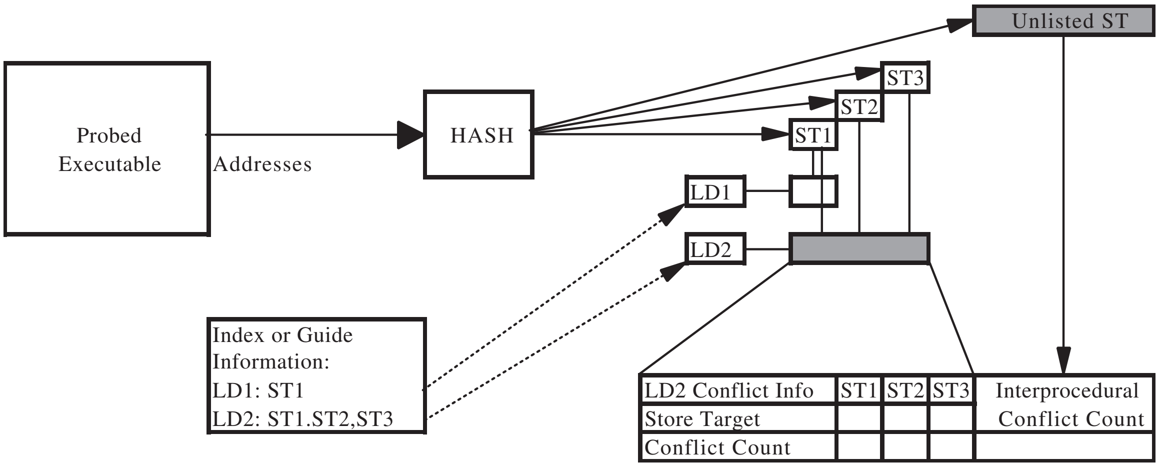

Figure 4.7: The *Sync Arc* Companion Approach

Arcs discussed in Chapter 2 in several ways. First, the *Sync Arcs* within a program are used as the set of interesting load-store pairs to profile, using a mechanism similar to index-based approach. By profiling *Sync Arcs*, information can be gathered to distinguish the arcs that exist due to conservative memory analysis from true dependences. *Sync Arcs* are also used because the resulting profile information can be embedded within the arc information in the intermediate representation of programs. Furthermore, *Sync Arcs* are already maintained throughout transformations in the IMPACT compiler, and thus profile information is preserved.

The *Sync Arc Companion* profile facility uses a combination of the index and address-based profiling approaches. It is still necessary for the addresses of all store operations to be entered into the global hash table. However, only the addresses of load operations selected prior to profiling are entered into the global hash table. The hash table entries are identical to the address-based approach, with the addition of a field indicating the unique function id of the last store operation that modified the data. When load operations occur, their target addresses traverse the hash table and find a matching entry. Then a comparison is made between the current function id and the function id within the entry. When the ids do not match, an interprocedural conflict has occurred, and is recorded. Interprocedural conflicts are counted so that information for the data speculative optimizations can be generated. This mechanism attempts to emulate the effectiveness of the region instrumentation approach for gathering interprocedural conflicts. However, the region instrumentation approach requires that the regions be determined prior to profiling, while the approach of this thesis attempts to provide the profile information without such additional processing. In order to track the interprocedural conflicts, the conflict structure for each selected load operation includes a count of interprocedural conflicts as well as counts of each individual store operation conflict.

Another benefit of profiling *Sync Arcs* within the IMPACT compiler is that it provides verification of the static memory dependence analysis and *Sync Arc* compiler support. In essence, *Sync Arcs* should represent all possible pairs of load and stores which may conflict. Thus, if memory profiling information exposes a conflict between a pair of load-store operations not represented by an arc, then potentially there is a missing *Sync Arc*. This beneficial side effect is another reason for integrating an effective memory dependence collection infrastructure into the compiler's environment.

A flow diagram of the related tools within the IMPACT compiler is shown in Figure 4.8. The module *Lguide* processes the *Sync Arcs* of a program and generates a *guide*

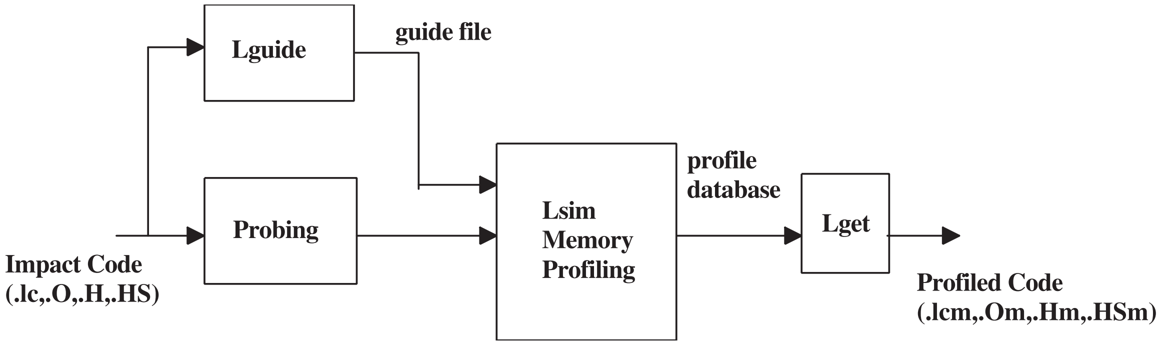

Figure 4.8: The *Sync Arc* Companion Profiling Path

file, which is the set of interesting load-store pairs to profile. The module *Lget* reads the profile information in the *profile database* generated during profiling, and annotates the information to the respective program operation. Overall, this figure illustrates how a program is first profiled and then annotated with PDC conflict information.

4.5.1 Profiling statistics

Several types of profiling statistics were generated for SPEC integer benchmarks and UNIX utilities using the *Sync Arc Companion* implementation. These statistics contribute insight into how each benchmark behaves in terms of memory dependences. For instance, Figure 4.9 illustrates the percentage of static and dynamic interprocedural memory conflicts, where the y-axis shows the percentage of conflicts that occur when one subroutine defines data at a memory location and a different subroutine subsequently uses the data from that memory location. Interprocedural memory conflicts mostly correspond to global variables and dynamically allocated memory. The benchmarks *026.compress*

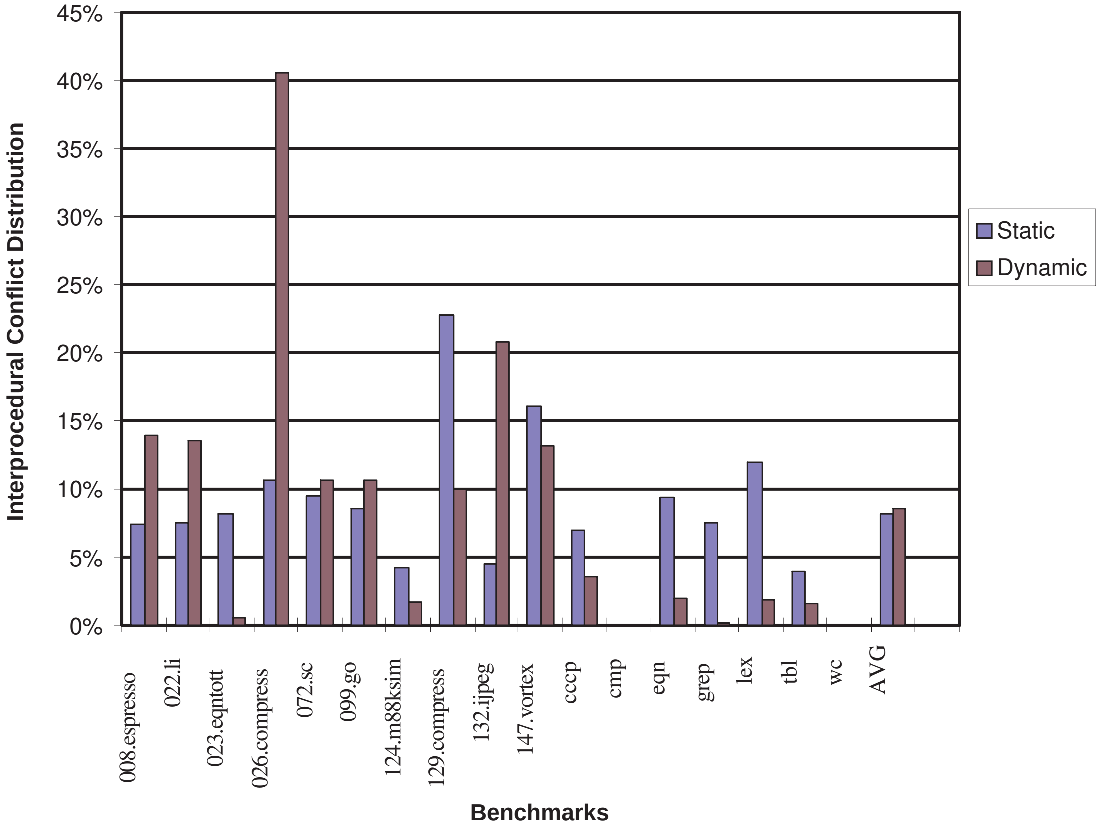

Figure 4.9: Static and Dynamic Interprocedural Memory Conflicts

and *132.ijpeg* have the most interesting behavior, since less than 11% of the static memory uses are interprocedural; but these benchmarks account for over 40% and 20% of the dynamic accesses, respectively. These conflict statistics include conflicts for load operations that retrieve data that was set in either caller or called functions, and thus is an overestimate of the number of opportunities for performing the load-subroutine advancement optimization.

Another useful statistic is the percentage of *Sync Arcs* within a program that memory profiling determines to have zero conflicts. Figure 4.10 shows this statistic for the benchmarks. The statistic of zero-conflict *Sync Arcs* can be misleading, since the memory reference profile information is limited to sections of the program exercised by the chosen profiling input set. Nevertheless, this information provides valuable feedback on the state of static memory disambiguation analysis, and on where to direct further analysis work. One interesting use of profile information is to remove zero-conflict rate arcs from a program, recompile, and then run the program again with the identical profile input. Without the zero-conflict arcs, a compiler's optimization and scheduling functions may be able to improve the program performance. This action is not a feasible compiler optimization, since other inputs could result in incorrect execution of the program. However, it would provide insight into the upper bound of performance achievable for certain programs.

One final statistic is the distribution of non-zero conflict rates within *Sync Arcs* displayed in Figure 4.11. The y-axis in Figure 4.11 represents the distribution of five non-zero conflict ranges (1-20%, 21-40%, 41-60%, 61-80%, and 81-100%) relative to all non-zero conflicts. Thus, long segments indicate a high occurrence of that conflict rate range within the program. Benchmarks such as *wc* and *cmp* are characterized by dependence arcs which have either high or low conflict rates. However, *099.go* behaves much differently, since 30% of its arcs have conflict rates between 20% and 80%. Overall, this figure illustrates that there are distinct classes of dependences that should be removed by using data speculation.

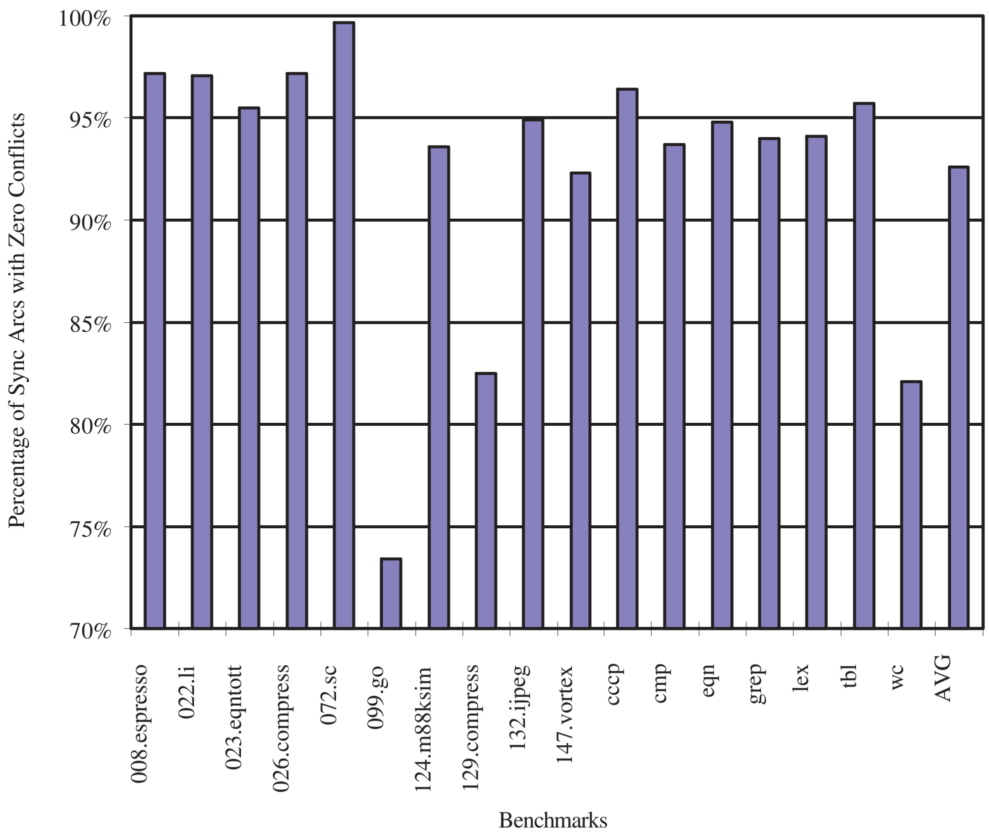

Figure 4.10: Zero Conflict *Sync Arcs* Percentages

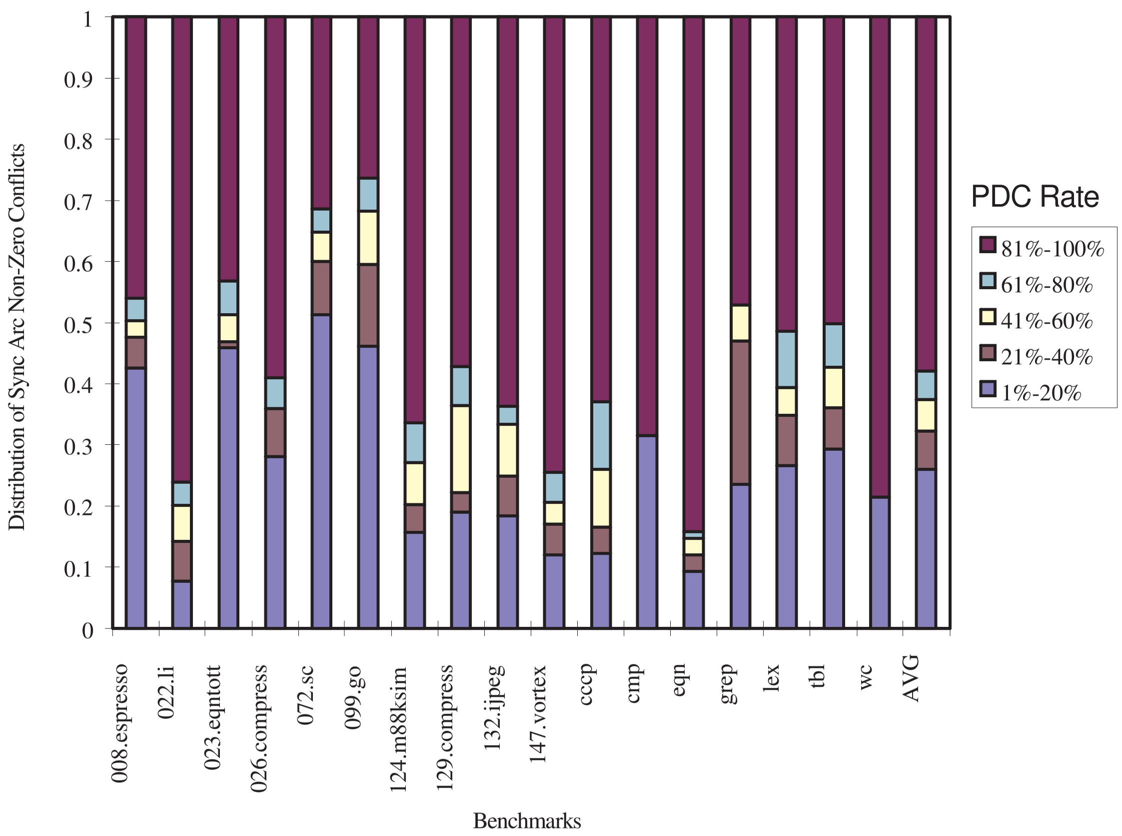

Figure 4.11: Distribution of Conflicts

5. EXPERIMENTAL PERFORMANCE EVALUATION

In this chapter, experimental results are presented from applying the previously discussed data speculative algorithms to a set of integer benchmarks consisting of the SPEC92, SPEC95, and common Unix benchmarks.

As mentioned in Chapter 2, the IMPACT simulator models an architecture's pipeline configuration, instruction and data caches, branch target buffer (BTB), and resource constraints. This allows the simulator to accurately compute the number of cycles required to execute a program for a simulated processor. Table 5.1 describes the architecture modeled for the data speculation experiments, and Table 5.2 shows the operation latencies used. The MCB model parameters used for the experiments were based on the best physical configuration presented in previous work [2]. Table 5.3 describes these parameters.

Table 5.1: Description of Simulated Architecture

Architecture Description
8-issue in-order execution superscalar processor
Extended version of HP PA-RISC instruction set
- Extensions for MCB
- Silent versions of all trapping instructions
64 integer, 64 floating-point registers
Dcache: 64k, direct mapped, non-blocking, 64 byte blocks,
8 cycle miss penalty, write-thru, no write allocate
Icache: 64k, direct mapped, non-blocking, 64 byte blocks,
8 cycle miss penalty
BTB: 1k entries, direct mapped, 2-bit counter,
2 cycle misprediction penalty

Table 5.2: Description of Simulated Architecture Latencies

Function	Lat	Function	Lat
Int ALU	1	FP ALU	2
(pre)load	2	FP multiply	2
store	1	FP div(SGL)	8
branch (check)	1	FP div(DBL)	15

Table 5.3: Description of Simulated MCB Model

MCB Description
64 entries, 8-way associative
5 address checksum width
2 cycle conflict penalty

5.1 Data Speculative Scheduling Performance Results

Based on the eight-issue architecture model specified in Table 5.1, scheduling estimates were made on three scheduling methods, superblock (BASE), MCB, and advanced MCB (AMCB). The MCB scheduling model uses the existing IMPACT data speculative scheduling algorithm described in Section 3.2, while the advanced MCB model uses the profile-driven techniques introduced in this thesis. Figure 5.1 illustrates the estimated speedup of the MCB and AMCB scheduling models relative to the BASE scheduling model. These estimates do not include approximations for the conflict rate of data speculative operations, and thus simply indicate an upper bound on the performance of each approach. It is clear from Figure 5.1 that the AMCB model uses more effective data speculative operations because it has significantly better estimated performance speedup than the MCB model.

Figure 5.2 shows simulation results for the MCB and AMCB models. Unlike the estimated schedule performance in Figure 5.1, the results of Figure 5.2 include the performance penalties associated with data speculative conflicts and correction code. Tables 5.4 and 5.5 list the conflict statistics for the MCB and AMCB experiments respectively. Although Figure 5.2 shows that the MCB scheduled benchmarks provide modest speedup for many benchmarks, the conflict statistics of Table 5.4 illustrate that the percentage of time that correction code is executed can be as high as 6%-12%. The performance loss due to the execution of correction code is part of the difference between the estimated performance speedup of Figure 5.1 and simulated performance of Figure 5.2.

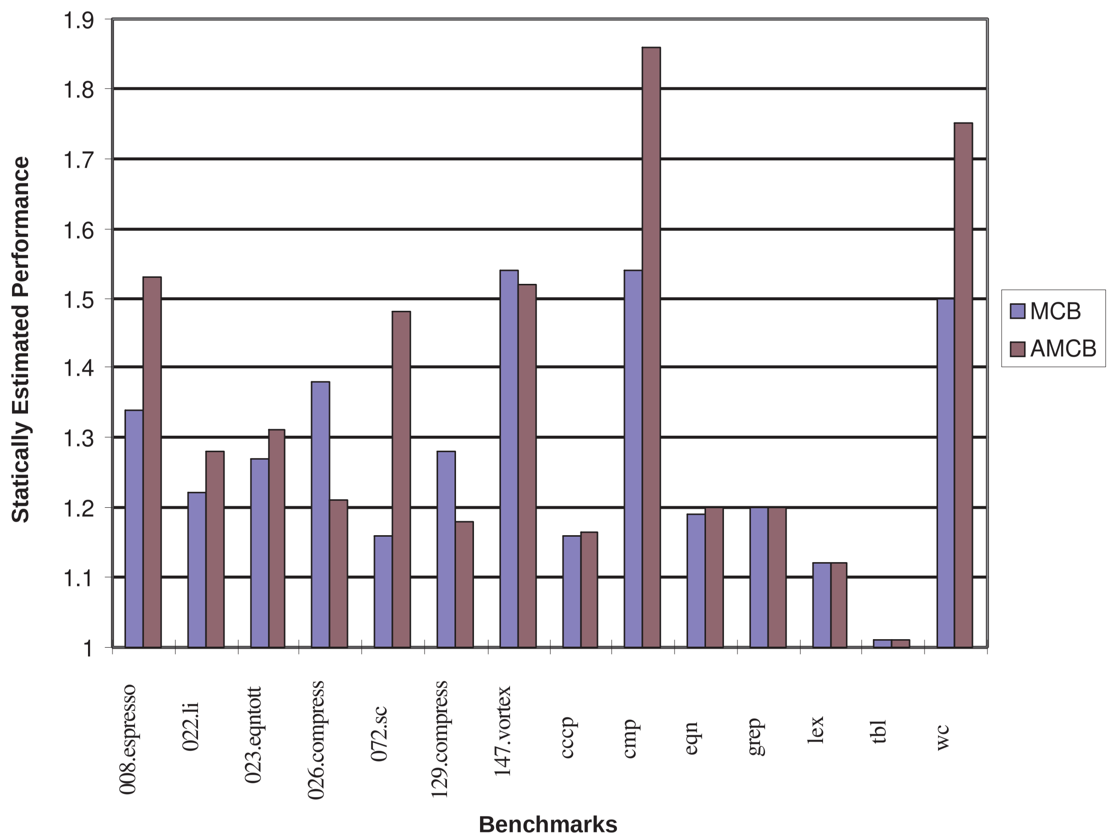

Figure 5.1: Estimated Performance of Data Speculative Scheduling Algorithms

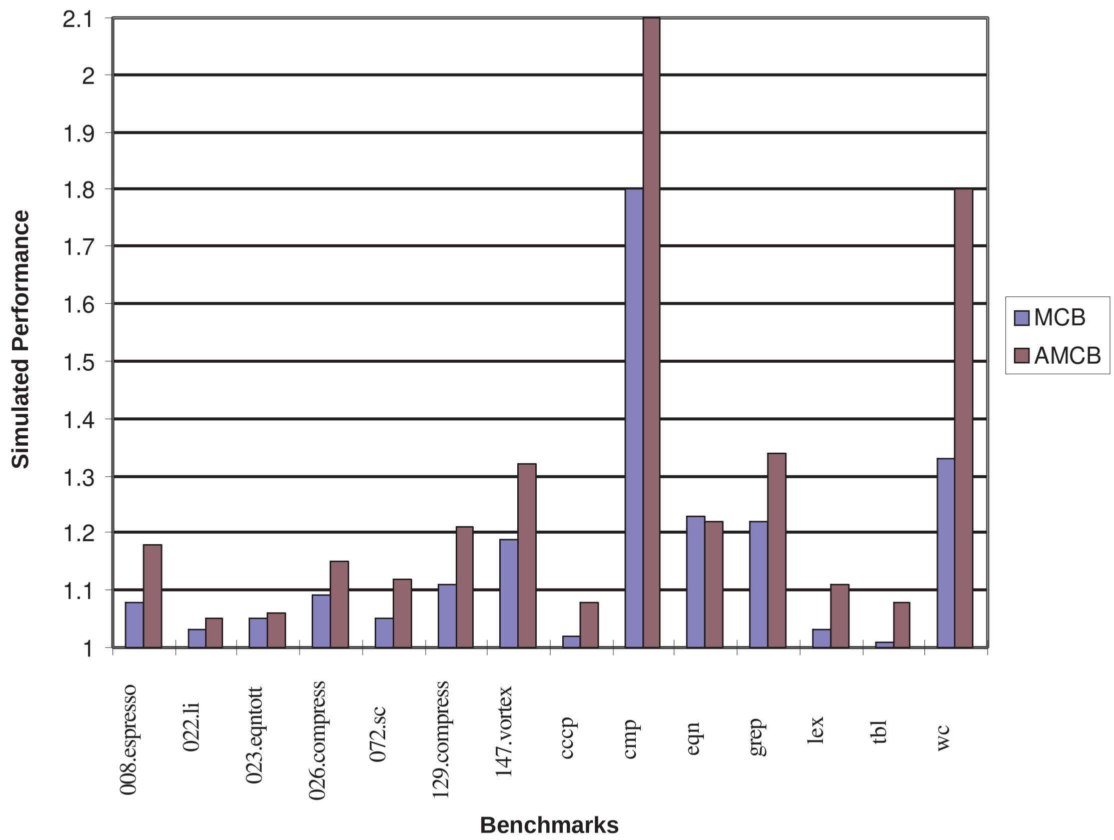

Figure 5.2: Simulated Performance of Data Speculative Scheduling Algorithms

Table 5.4: MCB Conflict Statistics (Old MCB Scheduling Approach)

Benchmark	Static Checks	Dynamic Checks	Total Conflicts	True Conflicts	% Checks Taken
008.espresso	335	5111K	281K	252K	5.55
022.li	31	273K	3.1K	2.7K	1.12
023.eqntott	47	70K	812	0	1.16
026.compress	44	494K	2.4K	2.04K	.49
072.sc	71	158K	1.7K	395	1.1
129.compress	39	499K	26.5K	1.5K	5.3
147.vortex	990	8600K	576K	312K	6.7
cccp	11	8K	16	0	.20
cmp	241	526K	21.6K	212	4.1
eqn	8	911K	3K	0	.32
grep	9	96K	499	0	.52
lex	31	81K	786	0	.97
tbl	4	5K	180	0	3.6
wc	71	125K	125	0	.10

Table 5.6 includes four columns not included in Table 5.4, *Static Load Verifies*, *Dynamic Load Verifies*,*Load Verify Conflicts*, and *% Reloads*. These columns present the conflict statistics for load verify (LDV) operations in the AMCB model, which are not generated in the MCB model.

Together Table 5.5 and Figure 5.2 illustrate several interesting results for the profile-driven data speculative algorithm approach. First, the percentage of checks taken has decreased significantly. In fact, the percentage of checks taken could be reduced to zero if the original algorithm only removed zero-conflict dependences during scheduling. In addition to the performance gained by reducing the number of conflicts, Table 5.5 shows that both the static and dynamic number of data speculative checks for several

Table 5.5: MCB Conflict Statistics (Advanced MCB Scheduling Approach)

Benchmark	Static Checks	Dynamic Checks	Total Conflicts	True Conflicts	% Checks Taken
008.espresso	519	6690K	85.6K	11K	1.28
022.li	19	209K	585	112	.28
023.eqntott	44	125K	725	0	.58
026.compress	14	235K	540	213	.23
072.sc	14	25K	63	0	.25
129.compress	13	131K	1703	156	1.3
147.vortex	398	6490K	142K	78K	2.2
cccp	4	5K	22	0	.44
cmp	99	640K	22.1K	0	3.43
eqn	3	1268K	4.2K	0	.033
grep	0	0K	0	0	0
lex	13	61K	802	0	1.31
tbl	3	3K	119	0	3.97
wc	34	215K	234	0	.01

benchmarks are dramatically higher compared to the results of Table 5.4. This is a result of the AMCB model including data speculative instances that the MCB model had statically estimated as not worth using.

Another interesting point displayed in Table 5.4 and 5.5 is that the MCB scheduling model uses more data speculative checks than the AMCB scheduling model in the *129.compress* and *147.vortex* benchmarks. Nevertheless, the AMCB scheduling model performs better since fewer checks are taken. The MCB scheduling model for these benchmarks experiences significant performance loss due to correction code being frequently executed. This loss indicates the importance of making intelligent decisions

Table 5.6: Load Verify Conflict Statistics (Advanced MCB Scheduling Approach)

Benchmark	Static Load Verifies	Dynamic Load Verifies	Load Verify Conflicts	% Reloads
008.espresso	90	852K	1618	.19
022.li	19	230K	966	.42
023.eqntott	9	42K	151	.36
026.compress	0	0K	0	0
072.sc	42	449K	1526	.34
129.compress	0	0K	0	0
147.vortex	829	4457K	5.8K	.13
cccp	6	8K	0	0
cmp	0	0K	0	0
eqn	1	96K	8.6K	.09
grep	9	55K	0	0
lex	4	2K	0	0
tbl	1	1K	0	0
wc	0	0K	0	0

when initiating data speculation. Overall, the benefit that profile information can provide is illustrated by the increased use of data speculation and the reduction of checks taken which result in improved performance speedup.

Finally, the conflict statistics of Table 5.5 indicate that scheduling algorithms that use memory reference profile information can virtually eliminate all of the true conflicts of data speculation. However, the profile information cannot eliminate false conflicts that result from the MCB implementation and design. False conflicts are due in part to overlapping uses of MCB entries during runtime. Scheduling algorithms which determine MCB resource contention may be able to reduce these conflicts.

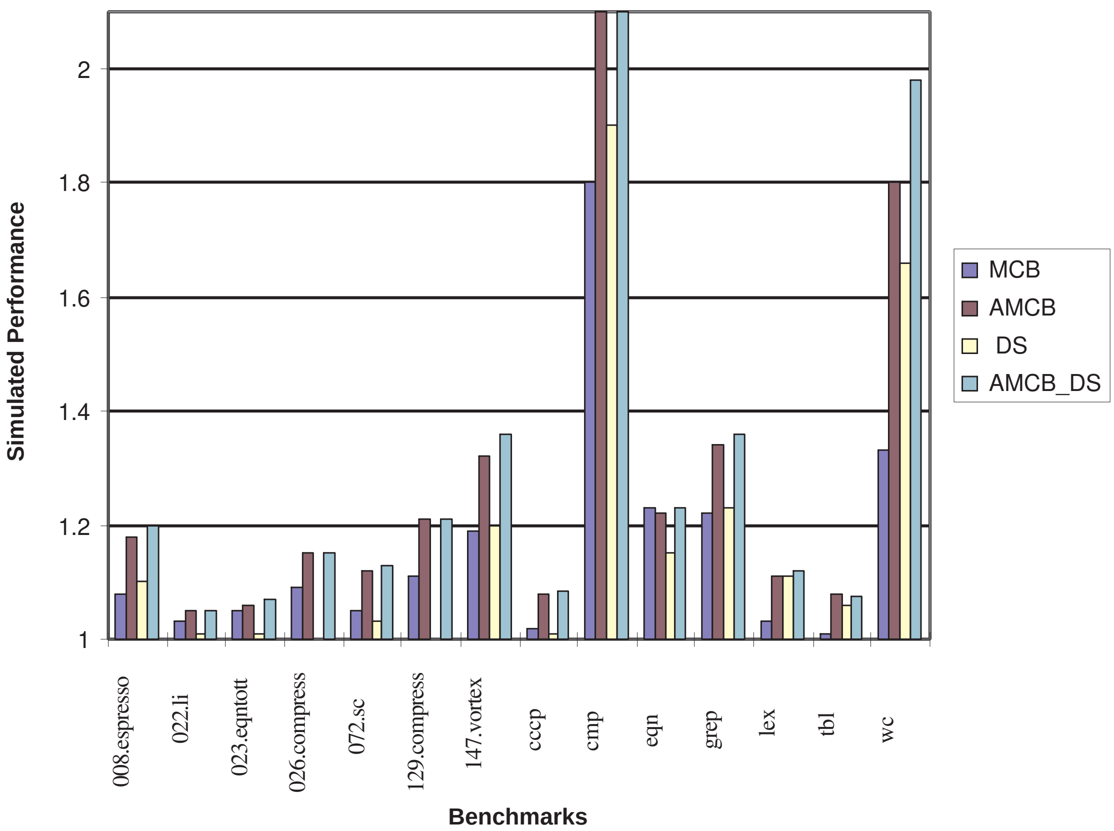

Figure 5.3: Simulated Performance of Data Speculation

5.2 Data Speculative Optimization and Scheduling Performance Results

The loop invariant data speculative algorithm developed in this thesis was also evaluated with the IMPACT simulator. Figure 5.3 shows the simulated performance of four data speculative approaches: MCB, AMCB, DS (data speculative loop invariant optimization), and AMCB with DS.

It is important to note in Figure 5.3 that the combined effect of using data speculative optimizations and scheduling algorithms is less than the product of the speedup when using either data speculative optimizations or scheduling algorithms separately. In many cases, data speculative optimizations may not complement the effects of data speculative scheduling. For instance, a load operation may be removed from a loop by optimization, thus removing an opportunity for the scheduler to perform data speculative scheduling within a control block. At the same time, some cases such as *cmp* seem to have results that illustrate a complementary nature between data speculative optimization and scheduling. In general, the results of Figure 5.3 indicate that data speculative optimizations can achieve a large performance speedup in the presence of data speculative scheduling. Overall, the results indicate that profile information can significantly enhance the effectiveness of both data speculative optimization and scheduling.

6. CONCLUSIONS AND FUTURE WORK

This thesis presents the implementation of data speculative optimizations and scheduling algorithms within the framework of the IMPACT compiler based upon memory profile information. In addition, it identifies different memory profiling techniques and demonstrates an effective implementation for the collection of memory profile information. Results show that by using Profile Data Conflict (PDC) rate information from memory profiling within the MCB scheduling approach, it is possible to greatly improve the performance of data speculation. Also, it has been shown that such information can consistently direct profitable data speculative optimizations such as loop invariant code removal. Overall, the use of memory reference profile information effectively identifies regions within a program where aggressive data speculation can be applied.

Future work in using memory profiling for directing data speculation will involve several areas of further investigation. The first part of this work will involve a comparison of the effectiveness of memory profiling and interprocedural analysis. In addition, work needs to be done to investigate how profile information can be used to create a set of static heuristics for applying profitable data speculation. Finally, an interesting question arises regarding how data profiling behaves compared to control profiling for different selections of input sets. Overall, the future work will investigate the importance of data speculation as new techniques emerge to further eliminate the bottlenecks caused by program control flow.

REFERENCES

[1] 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," in *Proceedings of the 25th International Symposium on Microarchitecture*, December 1992, pp. 45–54.

[2] D. M. Gallagher, "Memory disambiguation to facilitate instruction-level parallelism compilation," Ph.D. dissertation, University of Illinois, Urbana, IL, 1995.

[3] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu, "Dynamic memory disambiguation using the memory conflict buffer," in *Proceedings of the 6th International Conference on Architectual Support for Programming Languages and Operating Systems*, October 1994, pp. 183–193.

[4] W. W. Hwu et al., "The superblock: An effective structure for VLIW and superscalar compilation," Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, Tech. Rep. CRHC-92-23, February 1992.

[5] P. P. Chang, S. A. Mahlke, and W. W. Hwu, "Using profile information to assist classic code optimizations," Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, Tech. Rep. CRHC-91-12, April 1991.

[6] K. Subramanian, "Loop transformations for parallel compilers," Master's thesis, University of Illinois, Urbana, IL, 1993.

[7] B.-C. Cheng, "Pinline: A profile-driven automatic inliner for the impact compiler," Master's thesis, University of Illinois, Urbana, IL, 1997.

[8] G. E. Haab, "Data dependence analysis for Fortran programs in the IMPACT compiler," Master's thesis, University of Illinois, Urbana, IL, 1995.

[9] S. A. Mahlke, "Design and implementation of a portable global code optimizer," Master's thesis, University of Illinois, Urbana, IL, 1991.

[10] R. E. Hank, "Machine independent register allocation for the IMPACT-I C compiler," Master's thesis, University of Illinois, Urbana, IL, 1993.

[11] R. A. Bringmann, "Compiler-controlled speculation," Ph.D. dissertation, University of Illinois, Urbana, IL, 1995.

[12] P. P. Chang, D. M. Lavery, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, "The importance of prepass code scheduling for superscalar and superpipelined processors," *IEEE Transactions on Computers*, vol. 44, March 1995, pp. 353–370.

[13] S. A. Mahlke, W. Y. Chen, W. W. Hwu, B. R. Rau, and M. S. Schlansker, "Sentinel scheduling for superscalar and VLIW processors," in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992, pp. 238–247.

[14] N. J. Warter, "Modulo scheduling with isomorphic control transformations," Ph.D. Dissertation, University of Illinois, Urbana, IL, 1993.

[15] W. Y. Chen, "Data preload for superscalar and VLIW processors," Ph.D. Dissertation, University of Illinois, Urbana, IL, 1993.

[16] V. Kathail, M. S. Schlansker, and B. R. Rau, "HPL PlayDoh architecture specification: Version 1.0," Hewlett-Packard Laboratories, Palo Alto, CA, Tech. Rep. HPL-93-80, February 1994.

[17] J. C. Gyllenhaal, "A machine description language for compilation," Master's thesis, University of Illinois, Urbana, IL, 1994.

[18] W. Y. Chen, S. A. Mahlke, W. W. Hwu, T. Kiyohara, and P. P. Chang, "Tolerating data access latency with register preloading," in *Proceedings of the 6th International Conference on Supercomputing*, July 1992, pp. 223–236.

[19] Y. Wu, "Collecting memory reference conflict ratio for data speculation," Intel Corporation, Santa Clara, CA, Tech. Rep. MRL-96-23, June 1996.

[20] T. L. Johnson, "Automatic annotation of instructions with profiling information," Master's thesis, University of Illinois, Urbana, IL, 1995.