Dynamic Memory Disambiguation Using the Memory Conflict Buffer

David M. Gallagher William Y. Chen* Scott A. Mahlke John C. Gyllenhaal Wen-mei W. Hwu

Center for Reliable and High-Performance Computing
University of Illinois
Urbana-Champaign, IL 61801

Abstract

To exploit instruction level parallelism, compilers for VLIW and superscalar processors often employ static code scheduling. However, the available code reordering may be severely restricted due to ambiguous dependences between memory instructions. This paper introduces a simple hardware mechanism, referred to as the *memory conflict buffer*, which facilitates static code scheduling in the presence of memory store/load dependences. Correct program execution is ensured by the memory conflict buffer and repair code provided by the compiler. With this addition, significant speedup over an aggressive code scheduling model can be achieved for both non-numerical and numerical programs.

1 Introduction

A compile-time code scheduler improves the performance of VLIW and superscalar processors by exposing simultaneously executable instructions to the hardware. To be effective, the scheduler must be able to freely reorder instructions within the scheduling scope. Unfortunately, code reordering is often inhibited by *ambiguous memory dependences*, the situation where the relationship between a pair of memory references cannot be determined accurately at compile time. Because memory dependences often occur on program critical paths, such loss of code reordering opportunities can severely impair the effectiveness of code scheduling.

The problem of ambiguous memory dependences can be addressed by static dependence analysis, dynamic memory disambiguation, or some combination of the two. Static dependence analysis attempts to determine, at compile time, the relationship between pairs of memory references. In many cases, this analysis is able to determine the reference pairs are either definitely dependent or definitely independent, enabling the compiler to perform optimizations or code reordering. However, static analysis is limited in two situations: 1) when memory dependences are truly ambiguous at compile time; or 2) when the reference pairs are sometimes dependent during execution, e.g. only for some loop iterations. Additionally, accurate static analysis requires a significant compile-time investment and may be inappropriate for some applications.

Dynamic memory disambiguation resolves memory dependences during program execution. It is a viable alternative when sophisticated static analysis is not available, when rapid compile time is required, or when the application is dominated by references for which static analysis is ineffective. Both hardware and software approaches to dynamic disambiguation have been proposed. Out-of-order execution architectures allow the instruction issue unit to calculate run-time memory addresses and reorder memory operations based upon actual dependences [1]. However, the code reordering in these architectures is limited by the size of the visible instruction window.

A software-only approach to dynamic disambiguation is *run-time disambiguation* proposed by Nicolau [2]. By inserting explicit address comparison and conditional branch instructions, run-time memory disambiguation allows general code movement across ambiguous memory stores. The approach is illustrated in Figure 1. The original code segment in Figure 1(a) has two store operations followed by an ambiguous load. In Figure 1(b), the load has been moved above both stores, and explicit address comparison code has been added. With run-time disambiguation, however, the number of address comparison and conditional branch instructions inserted can be prohibitive as a result of aggressive code reordering: if m loads bypass n stores, $m \times n$ comparisons and

*Currently with Intel Corporation, Santa Clara, CA.

```
R1 = R2 * R3
M(R9+R10) = R11
M(R3+R7) = R1
R4 = M(R5+R8)
R6 = R4 + 1
```

a) Original Code

```
R1 = R2 * R3
R4 = M(R5+R8)
M(R9+R10) = R11
If (R5+R8 == R9 + R10)
     R4 = R11
M(R3+R7) = R1
If (R5+R8 == R3 + R7)
     R4 = R1
R6 = R4 + 1
```

b) Runtime Code

Figure 1: Runtime Memory Disambiguation Example.

branches would be required.

The Memory Conflict Buffer (MCB) scheme, first presented in Chen's thesis [3], extends the idea of run-time memory disambiguation by introducing a set of hardware features to eliminate the need for explicit address comparison instructions. The MCB approach involves the introduction of at most two new instructions: 1) *preload*, which performs a normal load operation, but signals the hardware that a possible dependence violation exists for this load [1]; and 2) *check*, which directs the hardware to determine if a violation has occurred and to branch to conflict correction code if required. Figure 2 demonstrates the MCB approach using the previous code example. In Figure 2(b), both the load and its dependent add have bypassed the ambiguous stores. Note the load is now a preload, and a check instruction has been inserted at the original location of the load. If the hardware determines an address conflict has occurred, the check instruction will branch to correction code, which re-executes the load and any dependent instructions. In contrast to run-time memory disambiguation, only one check operation is required regardless of the number of store instructions bypassed by the preload. As a result, the MCB scheme allows the compiler to perform aggressive code reordering with significantly less code expansion and execution overhead than run-time memory disambiguation.

2 Architectural Support

With the introduction of the preload and check opcodes, the compiler is free to move load instructions and their dependent operations past ambiguous stores. The MCB hardware supports such code reordering by 1) detecting the situation in which the ambiguous reference pair access the same location and 2) invoking a correction code sequence supplied by the compiler to restore the correctness of program execution. The situation where a preload and an ambiguous store access the same location will be referred to as a *conflict* between the reference pair. When this occurs, the reordered load and any dependent instructions which bypassed the store must be re-executed.

[1]The *preload* instruction is presented here to facilitate explanation of the MCB. We show in Section 4.3 that explicit *preload* opcodes are not necessarily required.

```
R1 = R2 * R3
M(R9+R10) = R11
M(R3+R7) = R1
R4 = M(R5+R8)
R6 = R4 + 1
```

a) Original Code

```
             R1 = R2 * R3
             R4 = M(R5+R8)   (preload)
             R6 = R4 + 1
             M(R9+R10) = R11
             M(R3+R7) = R1
             Check R4, Correction
Back:

Correction:  R4 = M(R5+R8)
             R6 = R4 + 1
             Jmp Back
```

b) MCB Code

Figure 2: Memory Conflict Buffer Example.

In order to detect conflicts as they occur, the MCB hardware maintains address information for each preload instruction as it is executed. The addresses of subsequent store instructions are then compared to this address information to determine whether a conflict has occurred. The hardware records the occurrence of the conflict; when the corresponding check instruction is encountered, the hardware performs a conditional branch to correction code if a conflict has been recorded. The correction code re-executes necessary instructions and then returns to normal program execution. In this section, we present MCB hardware to detect and record load-store conflicts and discuss other issues affecting the hardware.

2.1 MCB Design

The MCB hardware is responsible for storing preload address information for comparison to subsequent store addresses. Perhaps the most direct approach would be to store all address bits in some form of fully-associative structure. However, a fully-associative search of any reasonably-sized MCB implementation would likely impose constraints upon processor pipeline timing. Additionally, the hardware costs to record 32 or more bits of address information for each preload might prove prohibitive.

The MCB design presented in Figure 3 was developed with scalability, access time, and physical size constraints in mind. The MCB hardware consists of two primary structures, corresponding to the needs to store address information and to record conflicts which occur: 1) the *preload array*; and 2) the *conflict vector*.

The preload array is a set-associative structure similar in design to a cache. Each entry in the preload array

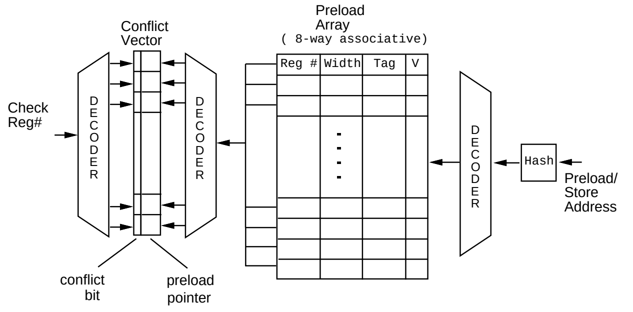

Figure 3: Set Associative MCB Design.

contains four fields: 1) the preload destination register number; 2) the preload access width; 3) an address *signature*; and 4) a *valid* bit indicating whether the entry currently contains valid data. The preload register field simply contains the register number of the preload destination. The address signature contains bits which contain a hashed version of the preload address. The access width field contains two bits to indicate whether the preload was of type character, half-word, word, or double word; additionally, this field contains the three least significant bits of the preload address. The use of the access width field will be discussed in a subsequent section.

The conflict vector is equal in length to the number of physical registers, with one entry corresponding to each register. Each entry contains two fields: the conflict bit and the preload pointer. The conflict bit is used to record that a conflict has occurred for a preload to this register. 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.

When a preload instruction is executed, the address of the preload is hashed to select which set in the preload array will store the preload. (The hardware to perform this hashing, as well as address signature generation, is detailed in the next section.) The preload array is set-associative; selecting an entry in which to store the preload information is identical to selecting an entry in a set-associative cache. If any entry within the set does not have its valid bit set, the preload information can be placed in this entry. When no invalid entry exists, a random replacement algorithm is used to select which entry to replace. If a valid entry is replaced, a *load-load conflict* has occurred; in this situation we can no longer provide safe disambiguation for the preload which is being removed from the array. We must therefore assume a conflict has occurred for this entry and set the conflict bit corresponding to the register number being removed. Note that for processors which support the execution of multiple preload instructions per cycle, the preload array must be multiported to allow simultaneous insertion of multiple preloads.

Having determined which entry in the preload array will be used for the current preload instruction, the destination register number and access width information are stored in the array. A second, independent hash of the preload address is performed to create the preload's address signature, which is stored in the signature field of the array. Unlike the tag field of a cache which must provide exact matching, this signature field can be hashed to reduce its size; the MCB can tolerate the occasional *false conflicts* which result from hashing. Simultaneous with storing the preload in the preload array, the conflict vector associated with the load's destination register is updated, resetting the conflict bit and establishing the pointer back to the preload array.

When a store instruction is executed, its address is hashed identically to the preload to determine the corresponding set in the preload array and to determine the store's address signature. The store's access width data (2 size bits and 3 LSBs) is also presented to the array. The store's signature and access width information are compared with the data stored within each entry of the selected set, to determine whether a conflict has occurred. For each entry in the set which is determined to conflict with the store, the conflict bit corresponding to the preload register is set; this requires that the conflict array be multiported to a degree equivalent to the associativity of the preload array. Two types of conflicts can arise when a store instruction is executed. If the load address and store address were identical or overlap, we say a *true conflict* has occurred. However, if the two addresses were different, and the conflict resulted from the hashing scheme used, we call this a *false load-store conflict*.

Thus, bits within the conflict vector can be set in one of three ways: 1) a true conflict; 2) a false load-store conflict resulting from the hashing scheme; or 3) a false load-load conflict resulting from exceeding the set-associativity of the preload array. Regardless of the source of the conflict, the hardware must assume it is valid and execute correction code to ensure program correctness. This is accomplished using the check instruction. The format for the check instruction is *check Rd, Label*, where *Rd* is a general purpose register number, and *Label* specifies the starting address of the correction code supplied by the compiler. When a check instruction is executed, the conflict bit corresponding to *Rd* is examined. If the conflict bit is set, the processor performs a branch to the correction code marked by *Label*. The correction code provides for re-execution of the preload and its dependent instructions. A branch instruction at the end of the correction code brings the execution back to the instruction immediately after the check, and normal execution resumes from this point.

The conflict bits are reset in two ways. First, a check instruction resets the conflict bit for register *Rd* as a side effect. Second, any preload that deposits a value into a general purpose register also resets the corresponding conflict bit. The valid bits within the preload array are reset upon execution of the corresponding check instruction, using the pointer within the conflict vector. Note that in the event the flow of control causes the check instruction not to be executed, the preload valid bits will remain set. However, this causes no performance impact because another preload of the destination register must occur before another check instruction can occur, resetting any spurious conflict.

Note that only preloads, stores, and checks need to access the address registers and the conflict vector. Accesses to the preload array are performed using the virtual address to avoid address translation delay. For store instructions, these accesses can be performed as soon as the store address is calculated; it is unnecessary to wait until the store data has been computed. For load instructions, MCB accesses are performed in parallel with the data cache access. Because the MCB is very similar to a cache in design and smaller than most caches, we believe it is unlikely that the MCB will affect the processor pipeline timing. However, further study of MCB timing is required within the context of a specific pipeline architecture.

2.2 MCB Address Hashing

Incoming preload and store addresses are used to select a corresponding set in the preload array. The most direct method to select one of n MCB lines is to simply decode $log_2 n$ bits of the address. However, testing revealed that this approach resulted in a higher rate of *load-load conflicts* than a baseline software hashing approach, most likely due to strided array access patterns causing additional conflicts. As a result, the MCB employs a permutation-based hardware hashing scheme.

Mathematically, our hardware hashing approach can be represented as a binary matrix multiplication problem, where matrix A is a non-singular matrix and $hash\_address = load\_address * A$. For example, consider the following 4x4 A matrix, used to hash 4-bit addresses:

```
1001
0010
1110
0101
```

To mathematically compute the hash address for incoming address 1011, we simply multiply this address by the matrix, obtaining hash address 0010. If matrix A is nonsingular, an effective hash of the incoming address is assured [4]. When mapping this scheme to hardware, each bit in the hash address is simply computed by XORing several of the incoming address bits, corresponding to the 1's in each column of the matrix. Thus $h3$, the most significant bit of the hash address, is the XOR of $a3$ and $a1$ of the incoming address; $h2$ is the XOR of $a1$ and $a0$, etc. This simple hardware scheme provides excellent hashing with only a small cost in time and hardware.

This same hashing approach is used to generate the address signature for incoming preload and store instructions. The signature is hashed in order to reduce the size of the MCB and to speed signature comparison. The signature is stored in the MCB for each preload, and is compared to the signature for incoming store instructions to determine if a *conflict* has occurred.

2.3 Handling Variable Access Sizes

Many instruction set architectures allow memory references to have byte addressability and variable access sizes. Thus, there arises the possibility that two memory references could access slightly different addresses, yet actually conflict. For example, the references:

```
store_int 0x40000000, R2
load_char R1, 0x40000001
```

represent a true load-store conflict. Although conflicts such as this are rare, they can occur in real code. An example where this might occur is in use of the *union* construct in *C*. To provide correctness, any code reordering scheme based upon memory disambiguation must account for the possibility of conflicts by memory operations with different access widths. One solution to this problem is to legislate it away; hardware designers can simply declare that accessing the same location with different width instructions is a poor programming practice and decide their hardware will not support it. A more general solution would require that any disambiguation technique provide adequate checks to ensure program correctness in the presence of variable width accesses.

To provide this capability, the MCB does not use the three LSBs of preload and store instructions when hashing to select the preload array line corresponding to the memory reference. Instead, these three bits, as well as two bits indicating the access size, are stored within the array for preload instructions. When a store occurs, its five bits are evaluated with the five bits stored for the preload to determine whether a conflict has truly occurred. A simple design for determining conflicts for these two five-bit vectors requires only seven gates and two levels of logic, assuming the architecture enforces aligned memory accesses.

a) Original Code	b) MCB Code
M (R1) = 7	R3 = M (R2)
R3 = M (R2)	R4 = R4 / R3
R4 = R4 / R3	M (R1) = 7
	Check R3, Correction

Figure 4: Speculative Execution of Excepting Instructions.

2.4 Handling Context Switches

Whenever a general purpose register needs to be saved to the memory due to context switches, neither the conflict vector nor the preload array must be saved. The only requirement is for the hardware to set all the conflict bits when the register contents are restored from memory. This simple scheme causes performance penalty only when the context switch occurs after a preload instruction has been executed but prior to the corresponding check instruction. Setting all conflict bits ensures all conflicts which were interrupted by the context switch are honored, but may cause some unnecessary invocations of correction code. The scheme also handles virtual address aliasing across multiple contexts. However, from our experience, this overhead is negligible for systems with context switch intervals of more than 100k instructions.

2.5 Speculative Execution

Speculative execution has been used in the literature to refer to executing an instruction before knowing that its execution is required. By moving an instruction above preceding conditional branches, an instruction is executed speculatively. In this paper, we extend the definition of speculative execution to refer to executing an instruction before knowing that it can be executed correctly. Instructions thus executed will be referred to as *speculative instructions*. In particular, a preload and its dependent instructions are executed before knowing if the preload will conflict with a store. The execution of these speculative instructions must be corrected if a conflict occurs.

There are two aspects of correcting the execution of speculative instructions. One is to correct the values generated by these instructions. The compiler algorithm described in Section 3 fulfills this requirement through intelligent compile-time renaming and register assignment. The more difficult aspect is to correctly handle exceptions. Since the value preloaded into the register may not be correct, there is a chance that a flow dependent instruction that uses the preload result may cause an exception. In the example in Figure 4, if $r1$ equals $r2$, then the value 7 is loaded into $r3$ in the original code segment. However, the value 0 may be preloaded into $r3$, in which case the divide instruction will cause an exception. Since the exception is due to an incorrect execution sequence, it must be ignored.

A solution is to provide architectural support to suppress the exceptions for speculative instructions [5]. A potential trap-causing instruction executed speculatively should be converted into the non-trapping version of the instruction. Therefore, the exception caused by the divide instruction in the example above would be ignored. However, the exception should be reported if there is no conflict between the preload and the store. Several schemes for precise exception detection and recovery have been proposed [6] [7] [8].

3 Compiler Aspects

To take full advantage of the MCB hardware support, a compiler must be able to intelligently reorder ambiguous store/load pairs, insert check instructions, and generate correction code. The compiler must also take into account the side-effects of aggressive code reordering. For example, over-speculating preload instructions can significantly increase register pressure and could result in a loss of performance due to spilling. In this section, we discuss the algorithms built into the IMPACT C compiler for exploiting the MCB support.

3.1 Basic MCB Scheduling Algorithm

To expose sufficient instruction-level parallelism (ILP) to allow effective code scheduling, the compiler must be able to look beyond basic block boundaries. In the IMPACT compiler, basic blocks are coalesced to form *superblocks* [9], an extension of trace scheduling [10], which reflect the most frequently executed paths through the code. Superblocks have a single entrance (at the beginning of the superblock), but may have multiple side exits. The superblock is the basic structure for scheduling in the IMPACT compiler.

The basic MCB algorithm involves the following steps for each frequently executed superblock:

1. Build the dependence graph.
2. Add a check instruction immediately following each load instruction, inserting necessary dependences.
3. For each load, remove store/load memory dependences.
4. Schedule the superblock, removing any unnecessary check instructions.
5. Insert required correction code.

```
R1 = R2 * R3
M(R9+R10) = R11
M(R3+R7) = R1
R4 = M(R5+R8)
R6 = R4 + 1
M(R9) = 0
R20 = R10 + 1
```

a) Original Code

```
R1 = R2 * R3
M(R9+R10) = R11
M(R3+R7) = R1
R4 = M(R5+R8)
Check R4, -
R6 = R4 + 1
M(R9) = 0
R20 = R10 + 1
```

b) Dependencies to Check Instruction

```
            R1 = R2 * R3
            R4 = M(R5+R8)   (preload)
            R6 = R4 + 1
            M(R9+R10) = R11
            M(R3+R7) = R1
            Check R4, Correction
            M(R9) = 0
            R20 = R10 + 1

Correction: R4 = M(R5+R8)
            R6 = R4 + 1
            Jmp Tail_Dup

Tail_Dup:   M(R9) = 0
            R20 = R10 + 1
```

c) Tail Duplication Code

```
            R1 = R2 * R3
            R4 = M(R5+R8)   (preload)
            R6 = R4 + 1
            M(R9+R10) = R11
            M(R3+R7) = R1
            Check R4, Correction
Back:       M(R9) = 0
            R20 = R10 + 1

Correction: R4 = M(R5+R8)
            R6 = R4 + 1
            Jmp Back
```

d) Tail Duplication Deleted

Figure 5: MCB Code Compilation.

The initial preparations for code scheduling, including building the dependence graph, are unchanged by the MCB algorithm. After the dependence graph has been built, a check instruction is added after each load instruction in the superblock. The destination register of the load becomes the source operand of the check, making the check instruction flow dependent upon the load. Initially, the correction block of the check is not defined. During code scheduling, the check instruction must maintain correct dependences; thus, it must be dependent upon the load and also inherit some of the load's dependences. Because we want flow dependent instructions of the load to be able to bypass the check, the check inherits only memory and control dependences from the load. Dependences to the previous and subsequent branch instructions are also added to the check instruction to ensure it remains within the load's original basic block. Figures 5(a) and 5(b) show the code from our previous example, and the code with the check instruction and its dependences inserted.

The next step in MCB scheduling is to remove store/load dependences. For each load, the algorithm searches upward, removing any dependence arcs to store instructions not determined to have a definite dependency. Associated with each load, the algorithm maintains a list of store instructions whose dependence has been removed. The algorithm currently only removes dependences to stores which precede the load, i.e. only removes flow dependences. Although nothing prevents dependences to subsequent stores (anti-dependences) from being removed, experience has shown there is little or no benefit from removing these dependences. To limit over-speculation of loads, the algorithm limits the number of store/load dependences which can be removed for each load. If too many dependence arcs are removed, our greedy scheduling algorithm is likely to move the load far ahead of its initial position, needlessly increasing register pressure and the probability of false conflicts in the MCB. Additionally, the algorithm ensures dependences are formed between the load instruction and any subroutine call in the superblock, preventing loads from bypassing subroutine calls. Thus, no MCB information is valid across subroutine calls.

Next, the superblock is scheduled. Each time a load instruction is scheduled, the list of stores associated with the load is examined. If all stores on the list have already been scheduled, the load did not bypass any stores during scheduling, and the associated check instruction can be deleted. The flow dependency between the load and the check ensures the check cannot be scheduled prior to the load; thus deletion of the check (and removal of its dependences) does not impact instructions already scheduled. If the load is determined to have bypassed a store during scheduling, the load is converted to its preload form. In our current implementation, one check instruction is required for each preload instruction. However, multiple check instructions could potentially be coalesced to reduce the execution overhead and code expansion incurred by the potentially large number of checks. Because the check is a single-operand instruction, extra bits should be available to accommodate a mask field to specify a set of registers which are to be checked by this instruction. For example, if a register bank with 64 registers is partitioned into eight sets of eight registers each, the check instruction would use three bits to specify which bank was being checked, and eight bits to specify the register mask. The coalesced check would branch to conflict correction code, which would have to provide correct execution regardless of which preload instruction experienced a conflict. Further research is required to assess the usefulness of coalescing check instructions.

3.2 Inserting Conflict Correction Code

The compiler provides conflict correction code for each preload instruction. When a check instruction determines a conflict has occurred, it branches to the correction code. The correction code re-executes the preload instruction and all dependent instructions up to the point of the check. (In the infrequent case that the load has bypassed a single store, the correction code can replace the re-execution of the preload with a simple move from the store's source register. In fact, the move itself may become unnecessary via forward copy propagation.) The original load instruction will not be a preload within correction code (because its check has already occurred), but any dependent instructions which happened to be preloads must be re-executed as preloads. During insertion of correction code, the compiler must check for any anti-dependences which would over-write source operands, such that these operands would not be available for execution within correction code. If anti-dependences are detected, they are removed by virtual register renaming.

Because scheduling is performed on superblocks which do not allow side entrances, the correction code cannot jump back into the superblock after re-executing the required instructions. Instead, the correction code jumps to *tail duplication* code, which is simply a duplicate copy of all superblock instructions subsequent to the check instruction. This tail duplication code (Figure 5(c)) ensures all dependences and register live ranges are calculated correctly during register allocation and post-pass scheduling. Following post-pass scheduling, however, the superblock structure is no longer necessary to the compiler and the code can be restructured to allow jumps back into the superblock. At this point, all jumps within correction code are redirected to jump back into the superblock immediately following the check instruction, and all tail duplication code can be deleted. Thus, the tail duplication code is only a temporary tool used by the compiler to maintain correct dependences and live ranges during register allocation and post-pass scheduling, and is removed prior to final code generation (Figure 5(d)).

4 Experimental Evaluation

To evaluate the MCB approach, experiments were conducted on a set of twelve *C* benchmark programs, including programs from SPEC-CFP92, SPEC-CINT92, and common Unix utilities. The need for better memory disambiguation is first demonstrated, followed by a description of our simulation methodology and the MCB performance results.

4.1 Potential Speedup from Memory Disambiguation

This current work is based on the premise that techniques such as superblock formation and loop unrolling have reduced the impact of non-sequential operations, and thus have increased the importance of memory disambiguation in achieving ILP. To support this premise, our benchmark suite was evaluated, using the three different types of disambiguation: 1) no memory disambiguation was performed, i.e all memory operations were assumed to conflict; 2) our compiler's present static disambiguation; and 3) ideal disambiguation, where memory operations are assumed to be independent unless static analysis indicates they are definitely dependent. The static disambiguation used by our compiler is typical of the static analysis performed on intermediate code by current commercial compilers. The analysis is strictly intraprocedural and uses only information available within the intermediate code for its analysis, i.e. no source-level information is used to aid the analysis. It is designed to be fast and fully safe. Note that the ideal disambiguation model used in this experiment may result in incorrect code if dependent instructions are reordered; it is presented to demonstrate an upper bound on the speedup available from memory disambiguation.

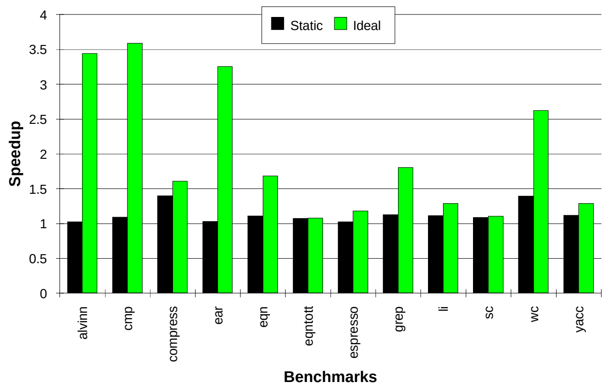

Figure 6: Impact of Memory Disambiguation on Code Scheduling.

For this experiment, an 8-issue architecture with uniform functional units is assumed. (A detailed description of the assumed architecture is included in the next section.) To estimate execution time, the code was profiled prior to scheduling to determine the execution frequency of each superblock. The code was then scheduled, using the various levels of disambiguation, to determine the number of cycles each superblock would take to execute. From this, an accurate estimate of required execution cycles can be determined, excluding cache effects and branch-misprediction penalties.

The results of this experiment are shown in Figure 6. The data reflects estimated execution time for the static and ideal cases, normalized to the no-disambiguation case. The speedup for the ideal case reflects only the effect from code scheduling; all other compiler modules (e.g classic optimizations) use static disambiguation. Only limited speedup is seen for the static case as the result of its inability to resolve many pointer accesses. Therefore, it is ineffective at achieving significant overlap between unrolled iterations of the inner loops in comparison to the ideal case. Overall, the speedup in the ideal case indicates ambiguous memory references are a significant impediment in a majority of the benchmarks evaluated.

4.2 Simulation Methodology

Unlike the experiment described in the previous section in which the resultant code could not be executed, all subsequent experiments were performed using a detailed emulation-driven simulation. Table 1 outlines the architecture modeled for these experiments (the *target* architecture). The instruction latencies used were those of the HP PA-RISC^{TM} 7100. The IMPACT simulator models in detail the architecture's prefetch and issue unit, instruction and data caches, branch target buffer (BTB), and hardware interlocks. This allows the simulator to accurately measure the number of cycles required to execute a program, as well as provide detailed analysis such as cache hit rates, BTB prediction accuracy, and total MCB true/false conflicts.

The compilation path required for simulation consists of several steps. Intermediate code is first run through the initial phase of our HP PA-RISC code generator, which transforms the code such that there is nearly a one-to-one correspondence between our intermediate form and HP assembly code. The code is then run through pre-pass scheduling, register allocation, and post-pass scheduling for the target architecture. MCB code is added during this stage of compilation. The output of this stage is the code which will actually be simulated. However, to create an executable file to drive the simulation, the functionality of the MCB must be emulated to allow the code to execute on the *host* architecture, an HP PA-RISC 7100 workstation.

```
            R1 = R2 * R3
            R4 = M(R5+R8)  (preload)
            R6 = R4 + 1
            M(R9+R10) = R11
            M(R3+R7) = R1
            Check R4, Correction
Back:

Correction: R4 = M(R5+R8)
            R6 = R4 + 1
            Jmp Back
```

a) Target Architecture Code

```
            R1 = R2 * R3
            R30 = R5 + R8
            R4 = M(R5+R8)
            R35 = 0
            R6 = R4 + 1
            R40 = R9 + R10
            M(R9+R10) = R11
            R45 = (R30 eq R40)
            R35 = R35 or R45
            R50 = R3 + R7
            M(R3+R7) = R1
            R55 = (R30 eq R50)
            R35 = R35 or R55
            Beq  (R35, 1), Correction
Back:

Correction: R4 = M(R5+R8)
            R6 = R4 + 1
            Jmp Back
```

b) Emulation Code

Figure 7: MCB Emulation Code.

Architectural Features
In-order execution superscalar processor
Uniform functional units (4 or 8)
Extended version of HP PA-RISC instruction set
- Extensions for MCB
- Silent versions of all trapping instructions
64 integer, 64 floating point registers
Dcache: 32k, direct mapped, blocking, 64 byte blocks,
12 cycle miss penalty, write-thru, no write allocate
Icache: 32k, direct mapped, blocking, 64 byte blocks,
12 cycle miss penalty
BTB: 1k entries, direct mapped, 2-bit counter,
2 cycle misprediction penalty
MCB support

Table 1: Simulated Architecture.

Following code scheduling, the code contains preload and check instructions, which are not executable by the host architecture. Thus, we must transform the code to accurately emulate MCB code. The MCB code is modified with explicit address comparisons similar to Nicolau's run-time memory disambiguation. Figure 7 illustrates the code changes required to emulate the MCB. To improve readability, the code is presented in pseudocode instead of HP PA-RISC assembly format. Register $R30$ holds the addresses of the preload, and $R40$ and $R50$ hold the addresses of the store. Registers $R45$ and $R55$ are set by an explicit comparison of the load address to the two store addresses. Because the preload instruction has bypassed numerous store instructions, $R35$ is used to record whether any of the stores caused a conflict. During emulation, the check instruction becomes a conditional branch based upon the value of $R35$.

Following insertion of emulation code, the code is probed to gather address and branch direction data for the simulation, and then the final phases of the code generation are performed to create an executable file. This executable file was run for all benchmarks and shown to produce correct results, verifying the correctness of the MCB code.

Simulation is performed on the target machine code, using probe data from the emulation path to determine actual run-time addresses and branch directions. The result is a highly accurate measure of the number of cycles required to execute the program on the target architecture. Due to the complexity of simulation, *sampling* [11] is used to reduce simulation time for large benchmarks. For sampled benchmarks, a minimum of

10 million instructions are simulated, with at least 50 uniformly distributed samples of 200,000 instructions each. Testing has shown sampling error to be less than 1% for all benchmarks.

4.3 MCB Evaluation

The performance of the MCB scheme was evaluated using the simulation technique described in the previous section. Evaluation was accomplished for both 4- and 8-issue processors, using various MCB configurations. Speedup in all figures is calculated as the ratio of MCB performance to the baseline architecture performance, i.e. a speedup of 1 indicates no performance gain. A set of twelve *C* benchmark programs was used, including programs from SPEC-CFP92, SPEC-CINT92, and common Unix utilities.

MCB Size and Associativity

The first MCB experiment was to measure MCB performance for various sizes of MCB. For this experiment, set associativity and signature field size were held constant (8-way set associativity and 5 signature bits) while the MCB size was varied from 16 to 128 entries, i.e. 2 to 16 sets. Additionally, performance for the perfect MCB case (i.e. false conflicts never occur) was measured to show asymptotic performance. Figure 8 shows the results from the six benchmarks evaluated. These six benchmarks were selected for this experiment because ambiguous memory dependences were shown to be major performance impediments for them in Figure 6. Speedup is shown for the MCB 8-issue architecture, relative to a baseline 8-issue architecture with no MCB. For most benchmarks, an MCB size of 32 or 64 entries was sufficient to approach perfect performance. The performance for *cmp* and *ear*, however, dropped significantly for sizes below 64 entries. This was the result of excessive load-load conflicts caused by several variables hashing to the same MCB location. Note that *cmp* did not show asymptotic performance even for an 128-entry MCB.

The results of MCB associativity testing are somewhat compiler-specific and are not shown. For most benchmarks, 8-way set associativity is required to achieve best MCB performance. Two factors heavily influence this need: 1) our compiler often unrolls loops up to 8 times; and 2) because the 3 LSBs of the load address are not used during hashing, up to 8 sequential single-byte loads will hash to the same MCB location. Thus, 8-way set associativity is needed to reduce the number of false load-load conflicts. Even at this associativity, the performance of *cmp* was impacted as a result of load-load conflicts caused by sequential loads and by independent variables hashing to the same location.

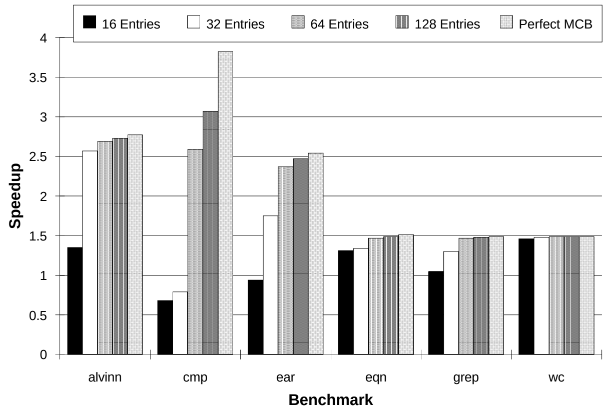

Figure 8: MCB Size Evaluation. Speedup of an 8-issue architecture for various size MCBs vs. an 8-issue architecture without MCB (8-way set-associative, 5 signature bits).

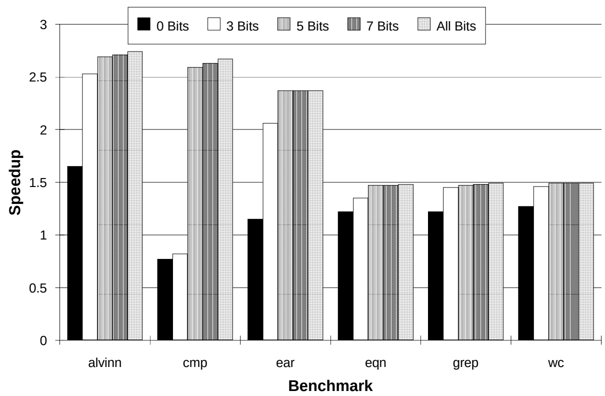

Figure 9: MCB Signature Size. Speedup of an 8-issue architecture with various size address signature fields vs. an 8-issue architecture without MCB (8-way set-associative, 5 signature bits).

Signature Field Size

To reduce the number of false load-store conflicts, the MCB contains a hashed signature field. The required width of this signature field was evaluated, holding MCB size constant at 64 entries, 8-way set associative. Performance was measured for field sizes of 0, 3, 5, and 7 bits, and performance for a full 32-bit signature is shown for comparison. MCB 8-issue speedup is again shown relative to the baseline architecture. Figure 9 shows the results; a signature size of 5 bits approached asymptotic performance of the full signature for all benchmarks.

MCB Performance

MCB performance for all twelve benchmarks was measured, using a 64 entry, 8-way set associative MCB with 5 signature bits. Figure 10 shows the performance for an 8-issue MCB architecture relative to the baseline 8-

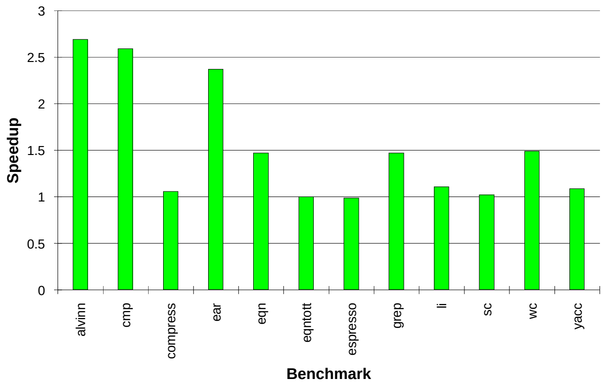

Figure 10: MCB 8-Issue Results. Speedup of an 8-issue architecture with MCB vs. without MCB (64 entries, 8-way set-associative, 5 signature bits).

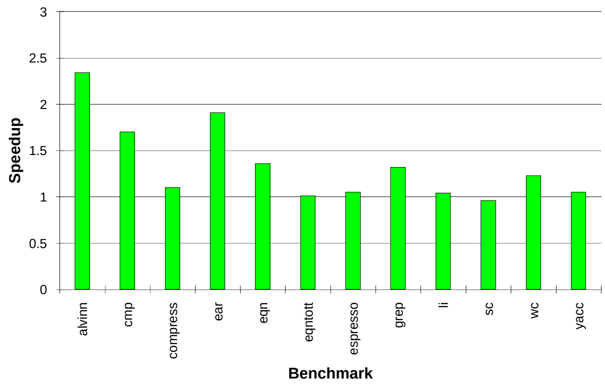

Figure 11: MCB 4-Issue Results. Speedup of a 4-issue architecture with MCB vs. without MCB (64 entries, 8-way set-associative, 5 signature bits).

issue architecture using our static disambiguation without MCB. MCB achieved significant speedup for six of the twelve benchmarks. Note the correspondence to the upper bound for speedup from Figure 6; MCB achieved good speedup for all benchmarks for which memory disambiguation was a significant impediment to ILP. Note also that the speedups for the two numeric benchmarks from SPEC-CFP92, *alvinn* and *ear*, were among the best achieved. This result is not surprising since these benchmarks are dominated by array accesses which are relatively difficult to disambiguate using only information available within the intermediate code (i.e. without interprocedural analysis or source-level information). Benchmarks such as *sc* and *eqntott* essentially achieved no speedup because the inner loops do not contain any store operations. For several benchmarks, including *compress* and *espresso*, MCB performance gains were somewhat masked by cache effects. In experiments using a perfect cache, *compress* achieved a 12% speedup and *espresso* achieved 7%. MCB code suffers slightly worse from cache effects because it experiences a greater overall number of cache misses. This increase in cache misses is because MCB's greater scheduling freedom allows more speculative execution of loads above branches; load misses from these speculative loads would not be experienced in less aggressively scheduled code.

Figure 11 shows the performance of a 4-issue MCB architecture relative to a baseline 4-issue architecture. As expected, performance gains are less than the 8-issue case; however, MCB still achieves moderate speedup for benchmarks for which memory disambiguation was significant. Note the performance of *sc* actually degraded on the 4-issue MCB architecture, as the result of increased data cache misses due to increased speculation of loads above branches.

Table 2 shows the conflict statistics for the 8-issue MCB architecture, using the same MCB configuration as Figure 10. The second column shows the total dynamic check instructions executed, followed by the number of true conflicts, false load-load conflicts, and false load-store conflicts. The final column shows the percentage of dynamic check instructions which branched to correction code. For most benchmarks, the percentage of time the correction code is executed is very low; the exception is *espresso*, which suffered from a large number of true conflicts. Note for all benchmarks except *eqn* and *espresso*, false conflicts were the primary cause of taken checks.

Table 3 shows the effect of the MCB compiler techniques on static and dynamic code size, again using an 8-issue architecture with a 64-entry, 8-way set-associative, 5 signature bit configuration. The addition of MCB code increased the static code size an average of 15.7% across the benchmarks. The benchmarks which showed the worst static code expansion were the very small benchmarks, in which the addition of a small number of check instructions and correction code to the most-frequently executed blocks made a significant change in the static code size. Note that the addition of check instructions resulted in a significant increase in the dynamic number of instructions executed for most benchmarks. However, the greater scheduling freedom allowed by MCB was in general able to pack this increased number of instructions into a tighter schedule and achieve speedup for many of the benchmarks.

Evaluating The Need for Preload Opcodes

The MCB approach adds the check instruction and preload versions of all load instructions. Because of the need to minimize the introduction of new opcodes, in this experiment the MCB approach was evaluated using no special preload opcodes. For this experiment,

Benchmark	Total Checks	True Confs	False Ld-Ld Confs	False Ld-St Confs	% Checks Taken
alvinn	802M	0	1708K	2374K	0.51
cmp	1023K	0	6632	1004	0.75
compress	2881K	28	248	13.3K	0.47
ear	2174M	0	14.2M	11.1M	1.17
eqn	4653K	43.2K	42.0K	3362	1.90
eqntott	4178K	0	11.8K	4356	0.39
espresso	11.5M	323K	94.7K	32.7K	3.93
grep	96.3K	0	0	501	0.52
li	1778K	0	112	11.3K	0.64
sc	301K	0	0	967	0.30
wc	321K	0	0	440	0.14
yacc	11.0M	11.5K	95.7K	1230	0.98

Table 2: MCB Conflict Statistics (8-issue, 64 entries, 8-way set-associative, 5 signature bits).

Benchmark	% Static Instruction Increase	% Dynamic Instruction Increase
alvinn	40.3	23.9
cmp	48.5	38.9
compress	13.6	7.0
ear	10.6	22.2
eqn	5.8	10.9
eqntott	12.8	0.2
espresso	6.5	8.5
grep	2.7	10.0
li	6.2	7.2
sc	1.1	1.7
wc	30.6	16.8
yacc	9.3	3.8

Table 3: MCB Static and Dynamic Code Size (8-issue, 64 entries, 8-way set-associative, 5 signature bits).

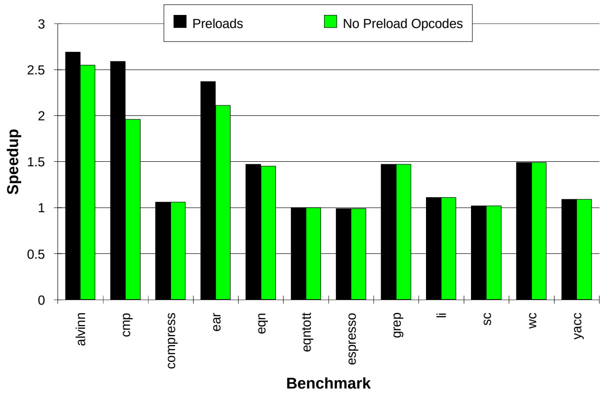

Figure 12: Impact of No Preload Opcodes. Speedup of an 8-issue MCB architecture with preload opcodes vs. the speedup of the same architecture without preload opcodes (64 entries, 8-way set-associative, 5 signature bits).

loads which have been moved above ambiguous stores are given no special annotation. The MCB design was modified such that all load instructions are processed by the MCB. Performance was evaluated for an 8-issue architecture, using a 64-entry MCB, 8-way set associative, with 5 signature bits.

Figure 12 shows the results of sending all load operations to the MCB. The graph compares the speedup achieved using preload instructions to the speedup when all loads are considered preloads. In both cases, speedup is calculated relative to the baseline 8-issue architecture without MCB support. For most benchmarks, only minimal performance degradation was experienced due to the absence of special preload opcodes. As seen in earlier experiments, *cmp* heavily tasks the MCB, and sending the additional non-preload instructions to the MCB increases the number of load-load conflicts, decreasing performance. In general, results indicate special preload opcodes are not required for MCB to achieve significant speedups.

5 Related Work

A great deal of research effort has been applied to static dependence analysis. Dependence analysis for arrays has reached a fair level of maturity for array references with linear subscripts [12] [13] [14] [15] [16]. Dependence analysis for pointers and recursive data structures is less mature, but is currently receiving a lot of attention [17] [18] [19]. However, static dependence analysis substantially increases compile time and may not be able to obtain exact dependence relations in all situations.

Dynamic memory disambiguation has received significantly less attention than static analysis. Dietz and Chi have proposed a combined hardware and compiler solution to avoid the overhead caused by ambiguous dependence relations [20]. Their approach was subsequently extended by Heggy and Soffa [21]. A set of aliasing registers called *CRegs* is used to store both the data and its corresponding memory address. The CRegs approach was envisioned to allow the elimination of redundant loads, despite the presence of intervening ambiguous stores. However, this approach might be extended to reorder ambiguous reference pairs during code scheduling. One limitation of this approach is the more complicated register file design, which may impact register access time.

Preload register update allows memory loads to be moved above any number of ambiguous stores, in order to better tolerate first-level cache hit latency [22]. Rather than trying to allow potentially aliased variables to simultaneously reside in registers as in the CRegs approach, preload register update is primarily designed to support compile-time reordering of ambiguous reference pairs. Simple address matching hardware is used to update the destination registers of the memory loads after code reordering.

A major limitation of the two compile-time code reordering approaches discussed above is that dependent instructions of a load may not be scheduled above any ambiguous stores, severely restricting code motion. Huang *et al* have proposed *speculative disambiguation* [23], to allow aggressive code reordering using predicated instructions. Similar to run-time disambiguation, it employs compiler techniques which allows both a load and its dependent instructions to bypass an ambiguous store. The method also allows two ambiguous stores to be reordered. In contrast to speculative disambiguation, the MCB approach does not require predicated execution support.

6 Conclusion

A great deal of research has been focused toward reducing the impact of control transfer instructions on instruction level parallelism. However, the success of those efforts has exposed ambiguous memory dependences as a significant impediment to exploiting ILP. This paper proposes a combined hardware and compiler approach, the memory conflict buffer (MCB), which performs dynamic memory disambiguation. The MCB allows aggressive code scheduling in the presence of ambiguous memory dependences. This is achieved by the compiler removing the dependences between ambiguous store/load pairs, allowing a memory load and its dependent instructions to be moved above any number of memory stores. The MCB hardware supports such code reordering by detecting situations when the ambiguous reference pair access the same location, and subsequently invoking a correction code sequence supplied by the compiler.

In a detailed simulation, MCB was shown to obtain substantial speedup for six of the twelve benchmarks evaluated. The MCB, or any other memory disambiguation approach, is not a panacea which will provide speedup for all programs. For some programs, control transfer instructions remain the primary bottleneck, and ambiguous dependences are not a significant problem. However, test results demonstrate that MCB provides substantial speedup for those programs whose ILP is limited by ambiguous memory dependences. One particularly significant result is that the MCB approach provides substantial speedups even if preload versions of load instructions are not provided. In this case, only one new instruction, the check, is necessary to support MCB.

This paper addresses the application of the MCB to code scheduling. We anticipate, however, that the MCB could also be successfully applied to the area of code optimization. For example, loop-invariant load or store removal is often hindered by ambiguous memory operations, and redundant load elimination may be prevented by ambiguous stores. We are currently studying the application of MCB to these problems and anticipate these opportunities for optimization will result in additional code speedup.

Acknowledgements

The authors would like to thank Roger Bringmann, Richard Hank, and Grant Haab, along with all members of the IMPACT research group for their comments and suggestions. Special thanks Pohua Chang at Intel Corporation, Tokuzo Kiyohara at Matsushita Electric Industrial Co., Ltd., and to the referees whose comments and suggestions helped to improve the quality of this paper significantly. This research has been supported by the National Science Foundation (NSF) under grant MIP-9308013, Joint Services Engineering Programs (JSEP) under Contract N00014-90-J-1270, Intel Corporation, the AMD 29K Advanced Processor Development Division, Hewlett-Packard, SUN Microsystems, NCR and the National Aeronautics and Space Administration (NASA) under Contract NASA NAG 1-613 in cooperation with the Illinois Computer laboratory for Aerospace Systems and Software (ICLASS). John Gyllenhaal was also supported by an NSF Graduate Research Fellowship.

References

[1] P. P. Chang, W. Y. Chen, S. A. Mahlke, and W. W. Hwu, "Comparing static and dynamic code scheduling for multiple-instruction-issue processors," in *Proceedings of the 24th An-*

nual International Symposium on Microarchitecture, pp. 25–33, November 1991.

[2] A. Nicolau, "Run-time disambiguation: coping with statically unpredictable dependencies," *IEEE Transactions on Computers*, vol. 38, pp. 663–678, May 1989.

[3] W. Y. Chen, *Data Preload for Superscalar and VLIW Processors*. PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1993.

[4] B. R. Rau, "Pseudo-randomly interleaved memory," in *Proceeding of 18th Internation Symposium on Computer Architecture*, pp. 74–83, May 1991.

[5] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman, "A VLIW architecture for a trace scheduling compiler," in *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 180–192, April 1987.

[6] 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*, pp. 238–247, October 1992.

[7] R. A. Bringmann, S. A. Mahlke, R. E. Hank, J. C. Gyllenhaal, and W. W. Hwu, "Speculative execution exception recovery using write-back suppression," in *Proceedings of 26th Annual International Symposium on Microarchitecture*, December 1993.

[8] M. D. Smith, M. S. Lam, and M. A. Horowitz, "Efficient superscalar performance through boosting," in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 248–259, October 1992.

[9] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The Superblock: An effective technique for VLIW and superscalar compilation," *Journal of Supercomputing*, vol. 7, pp. 229–248, January 1993.

[10] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, vol. c-30, pp. 478–490, July 1981.

[11] J. W. C. Fu and J. H. Patel, "How to simulate 100 billion references cheaply," Tech. Rep. CRHC-91-30, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, 1991.

[12] U. Banerjee, *Dependence Analysis for Supercomputing*. Boston, MA: Kluwer Academic Publishers, 1988.

[13] G. Goff, K. Kennedy, and C.-W. Tseng, "Practical dependence testing," in *Proc. 1991 SIGPLAN Symp. Compiler Construction*, pp. 15–29, June 1991.

[14] D. E. Mayden, J. L. Hennessy, and M. S. Lam, "Efficient and exact data dependence analysis," in *Proc. 1991 SIGPLAN Symp. Compiler Construction*, pp. 1–14, June 1991.

[15] W. Pugh and D. Wonnacott, "Eliminating false data dependences using the omega test," in *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 140–151, June 1992.

[16] E. Duesterwald, R. Gupta, and M. L. Soffa, "A practical data flow framework for array reference analysis," in *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pp. 68–77, June 1993.

[17] W. Landi and B. G. Ryder, "A safe approximate algorithm for interprocedural pointer aliasing," in *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 235–248, June 1992.

[18] A. Deutsch, "Interprocedural may-alias analysis for pointers: Beyond k-limiting," in *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pp. 230–241, June 1994.

[19] M. Emami, R. Ghiya, and L. J. Hendren, "Context-sensitive interprocedural points-to analysis in the presence of function pointers," in *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pp. 242–256, June 1994.

[20] H. Dietz and C. H. Chi, "Cregs: A new kind of memory for referencing arrays and pointers," in *Proceedings of Supercomputing '88*, pp. 360–367, Nov. 1988.

[21] B. Heggy and M. L. Soffa, "Architectural support for register allocation in the presence of aliasing," in *Proceedings of Supercomputing '90*, pp. 730–739, Nov. 1990.

[22] 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 1992 International Conference on Supercomputing*, pp. 104–113, July 1992.

[23] A. S. Huang, G. Slavenburg, and J. P. Shen, "Speculative disambiguation: A compilation technique for dynamic memory disambiguation," in *Proceedings of the 21st International Symposium on Computer Architecture*, pp. 200–210, April 1994.