Improving the Performance of Data Caches in Systems with Large Miss Latencies

Koray Oner and Michel Dubois

CENG Technical Report 92-14

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213)740-4475

IMPROVING THE PERFORMANCE OF DATA CACHES IN SYSTEMS WITH LARGE MISS LATENCIES

Koray Öner and Michel Dubois

University of Southern California

Department of EE-Systems

Los Angeles, CA90089-2562

Abstract

With current and projected processor technologies, memory accesses are quickly becoming a major bottleneck of modern computing systems. Even with a good cache, the miss penalty can be so high that the processor works at greatly reduced efficiency. Whereas stores can be buffered in a store buffer to hide store miss penalties, loads cannot be dealt with so easily because the processor needs the data returned by the load.

In this paper we introduce a simple processor/cache architecture with non-blocking loads. We then report results of trace-driven simulations of several FORTRAN DO-Loops. We first show that the architecture is ineffective unless loads can be hoisted away from the instructions that need the returned value. We then apply load hoisting to the loops and show the possible performance improvements for the systems with very large load miss latencies.

1 INTRODUCTION

With the improvements in VLSI technology and processor architecture, the gap between the speed of processors and the speed of memory systems made of DRAMs (Dynamic Random Access Memories) is widening. Observing the past and present trends we can say that this speed gap will probably continue to widen unless a major breakthrough occurs in memory technology. One very successful technique to deal with this technological reality is to access the main memory through very fast buffers called *cache memories*, which take advantage of the locality observed in most programs to serve memory requests faster. When a reference is not found in cache (miss), it must be loaded from the main memory. The time taken by a miss is called the *miss latency*. While the miss is being processed, the processor may be capable of executing different instructions, but it must eventually block when an instruction needs the returned value. The time during which the processor is blocked on a miss is called the *miss penalty*. Miss penalties affect the efficiency and the MIPS rate of the processor. Currently miss latencies are 10 to 50 cycles, but in the near future we can expect latencies in the range of 100 to 200 cycles, especially in multiprocessor systems.

In general, stores can be buffered in a store buffer to hide the store miss penalty. Load misses cannot be dealt with so easily because the processor must wait for the data on a load. In systems with large miss latencies loads must be non-blocking both in the processor and in the data cache so that processor execution can be overlapped with load misses. Caches that do not block the processor on a miss have been called *non-blocking* or *lockup-free* caches [4, 5, 6, 8, 11].

Other than lockup-free caches different techniques are proposed to solve the memory access latency problem. One of them is prefetching the data much before its usage. Therefore, data will be in the cache when it is needed by the load instruction. Prefetching can be done in software [4, 6, 10] or in hardware [13]. Prefetching will be explained in more detail in the next section.

Another method is called *Multi-threading*[23] in which processor does context switching between different instruction streams in case of a cache miss. In this method more than one register set should be present on the chip and the processor should be able to do context switching very fast. Therefore, multi-threading processors are much more complex. Moreover, having several running contexts decreases the hit rate of the cache.

The last method is observed in *Decoupled Access/Execute Architectures*[24, 25]. In these systems operand access and execution are decoupled and two (or more) processors process two (or more) separate instruction streams. These processors communicate via hardware queues which try to smooth the changes in data supply and request rates providing higher average transfer rate even in a system with low peak transfer rate capacity. In [25] two memory systems are examined: fast (Memory access time (Tm) = 1 cycle) and slow (Tm = 4 cycles). In our study we go beyond this and examine memory systems with memory access latencies up to 200 cycles.

While a load miss is being resolved by the cache, i.e. while the load miss is *pending* in the cache, the processor is free to execute instructions following the load for as long as these instructions do not need the data returned by the load. Depending on the cache architecture, more than one miss may be pending at any time. Since the hit ratio of instructions is usually very high and stores can be buffered in a store buffer, data load misses usually are the largest contributing factor

to the overall miss penalty. If misses can be totally overlapped with the execution of other instructions, then the system becomes tolerant to the miss latency in the sense that the latency does not affect the performance. Several recent papers have reported very encouraging results [6, 10] on the effectiveness of lockup-free caches. Nevertheless, very few implementations exist in commercial systems [1, 2, 3, 12]. One reason is the complexity of non-blocking caches. Non-blocking caches in commercial systems have support for a single pending miss, which is a serious limitation as we will see. In this paper we propose a non-blocking processor/cache architecture which eliminates the complexity of current designs and supports an unlimited number of pending misses.

To be effective, however, this architecture needs significant compiler help. Loads that miss in the cache must be moved away from the first instruction which uses its target register. In this paper, we examine the effectiveness of a basic code motion transformation to take advantage of non-blocking loads both in the processor and in the cache. Simple code motion transformations are applied to selected programs of the Lawrence Livermore Loops [9]. We first identify the features of the code in each Loop[1] which may hinder code motion. Then we classify the Lawrence Livermore Loops according to these features and we select five Loops. The selected Loops are compiled and the compiled code is transformed by hand to produce code with different amounts of code hoisting. Using an instruction-level SPARC simulator we then generate traces for the original code produced by the compiler and for the transformed code. These traces are used to drive a simulator for various miss latencies. Our results show that a simple load hoisting procedure can be very effective in the case of many loops. Based on the simulation results, refinements to both the basic load hoisting procedure and cache architecture are proposed to further improve Loop performance in the presence of large miss latencies.

In Sections 2 and 3, we describe the processor and cache architectures simulated in this study, as well as the simple code motion procedure applied to the Loops. The selection of five Loops is done in Section 4. In Section 5, the simulation results are presented and discussed. Finally, in Sections 6 and 7, we propose possible improvements and conclude.

2 LOCKUP-FREE CACHE ARCHITECTURES

Even small miss ratios are detrimental to the throughput of very fast processors. For example, if we consider a processor capable of executing one instruction per cycle when the cache always hits, the average time to execute an instruction is $1 + r \cdot T_m$, where T_m is the miss penalty and r is the average number of misses per instruction. If T_m is larger than 100 processor cycles, the efficiency of the processor is less than 50% even if r is 1%. This problem is even more severe for superscalar/superpipelined processors [12], in which the average number of instructions executed per cycle may be two or three. To offset the effects of the large miss penalties, the cache must be non-blocking or lockup-free.

The implementation of a lockup-free cache was first described in Kroft's paper [8]. In Kroft's design the cache controller stores information about misses in a set of fully associative registers, called Miss Status Holding Registers (MSHRs), and then sends the miss request to main

1. In the text Lawrence Livermore Loops are referred as 'Loop' while general DO-Loops are referred as 'loop'.

memory. Each MSHR contains the following information: cache buffer address, input request address, input identification tags (one per word), send-to-CPU indicators (one per word), in-input-stack indicators (one per word), partial write codes (one per word), number of words processed for that specific cache block, valid information, and obsolete indicator. Kroft makes a useful distinction between *primary* and *secondary* misses. A primary miss is the first pending miss for a cache block. Depending on the block size and the miss latency there may be several pending *secondary misses*. A secondary miss occurs on data blocks for which there is a primary pending miss; a secondary miss is most likely to be a hit in the case of a blocking cache and therefore should not trigger a memory request in a lockup-free cache. In his design, Kroft allocates one MSHR for each primary or secondary miss. The total number of pending misses is bounded by the total number of MSHRs. Kroft recommends to use at most four MSHRs based on his evaluations. His recommendation may be valid if no code transformation is applied. In our study we have observed at times six pending *primary* misses. When all MSHRs are busy the occurrence of a miss causes the cache to block. Therefore the number of MSHRs is a severe limitation to reducing the miss penalty.

Lockup-free caches were used before Kroft's paper; most IBM mainframes after the IBM 3033 have a lockup-free cache with one MSHR [3]. More recently, lockup-free caches were also proposed for the RP3 prototype in [1]. In [5], Kroft's design was simplified by storing the information about pending misses in the cache block itself. To be able to do this the authors added a new cache state called 'Transit' which means that a miss is pending for the cache block. Their design solves the limitations due to the number of MSHRs, but still requires saving extensive information about the miss; moreover it does not address the added complexity of the processor, which must support non-blocking loads. Non-blocking loads are implemented for example in the IBM 801 RISC processor [2] and in the IBM RS/6000 superscalar machine [12]. In these machines, when a load misses in the cache, the processor executes subsequent instructions in parallel with the load miss; later, when the load completes, a low-level trap must interrupt the processor and must direct it to load the value in a specific register. We call such non-blocking loads *in-register loads*, because they reserve a register to store the value returned by the load. Right after the completion of an in-register load, the data is both in cache and in register.

Non-blocking loads may also simply access the data without storing the value in a register. We call such loads *in-cache loads*. Right after the completion of an in-cache load the data is in cache. With in-cache loads, compiler-controlled prefetching can be done to reduce the miss rate. However, to be successful, this approach requires the overhead of prefetching to be low. This overhead comes from added execution: the prefetch load is an additional instruction and must be preceded by its address computation. Additionally, prefetch instructions increase memory traffic when data is prefetched unnecessarily. A prefetch is useless if the following in-register load for the same address would be a hit, or if the block is replaced between the prefetch and the corresponding in-register load. Techniques must be developed to minimize the number of generated prefetch instructions and to save the computed address of the prefetch load so that it can be reused by the corresponding in-register load. Porterfield's [10] proposed the idea of Overflow Iteration to predict loads that miss in the cache and to generate prefetch instructions for these loads only. He also suggested to use separate registers for storing the calculated addresses. So far, the performance studies of non-blocking caches has been confined to prefetching with in-cache loads [4, 6, 10] carefully hand-placed in the code. Recently, in his paper Baer [13] has proposed a hardware

scheme which does in-cache prefetching based on the prediction of the execution of the instruction stream. By contrast, in our paper, we rely exclusively on in-register loads and on a simple code hoisting procedure which can be performed by the compiler.

We now propose a combined processor/cache architecture, which practically eliminates the complexity traditionally associated with non-blocking in-register loads and non-blocking caches. An interesting feature of the architecture is the way in-register load misses are dealt with. When an in-register load misses in the cache, the address of the word is loaded in the register and the register is locked (using a one-bit tag such as the tag used to detect hazards on registers [7]). The subsequent instruction reading the data blocks the processor and the address in the register is then used to access the cache. The cache interacts with the processor as follows:

- Primary miss: a cache block frame is allocated (implying a possible replacement) and the state of the block is set to *pending*. A pending block cannot be replaced. If the access is an in-register load, the address is returned to the processor and stored in the register; the register is tagged as busy. In all cases, the block is loaded in cache from memory and neither the processor nor the cache are blocked while the miss is pending.
- When an instruction reads a busy register an *implicit* load is triggered using the address in the register. An implicit load always blocks the processor, the point being that the data is needed to complete the instruction execution.
- When the implicit load completes (whether it missed or not) the register is loaded with the data value and the register is unlocked.
- Secondary miss: a secondary miss is detected when the accessed block is pending in the cache and the tags of the memory request and the mapped block are the same. In the case of an in-register load the address is stored in the register and the register is tagged busy. No request is sent to memory. The cache and the processor remain unblocked.
- In the case of a conflict miss to a pending block the processor is blocked until the first pending access is completed. (A conflict miss occurs if the tags of the memory access and the mapped cache block are different.)

In this design, we do not need to keep track of pending loads and their target registers in MSHRs, which limited the number of pending misses in Kroft's design. Secondary store and load misses are handled automatically and load completion traps in the processor are eliminated.

3 CODE MOTION

In order to increase the overlap of load misses with execution, each load instruction must be hoisted in the code away from the following instructions which use its target registers. We define the *dependency distance* as the minimum number of processor cycles between the load and the first following instruction which reads the target register (either another load or a register-to-register instruction). In the code generated by current compilers, the dependency distances are very short (typically a few processor cycles) because current compilers generate code for architectures with blocking loads. Although the IBM RS6000 employs a lockup-free cache, our study of the assembly code generated by the RS6000's C compiler indicates that IBM does not perform code transformations to exploit it. Warren reports that such optimizations are possible but that IBM does not exploit them [12]. In most cases loads are issued just before register-to-register

instructions in order to reduce register consumption. By contrast, the primary goal of compilers for non-blocking cache/processor architectures should be to generate code with long dependency distances. In fact, since a hit in the cache does not waste processor cycles, the only loads to move are the ones that are likely to miss. Ideally compilers should try to predict the load misses and move these loads only, since moving *all* loads is wasteful of registers.

In the case of FORTRAN DO-loops load hoisting can be facilitated by pipelining the loads across iterations of the loop: memory values needed in iteration i are loaded in iteration i-k, k being the number of stages in the pipeline; once they have been pipelined the loads are hoisted as far up in the body of the loop as possible. We call this first technique *load pipelining*. Load pipelining is similar to a more general technique called *software pipelining* [14, 15, 16, 17, 18, 19], but it is limited to loads and is implemented at the intermediate code or assembly code levels. *Loop unrolling* at the source code level followed by load hoisting in the body of the unrolled loop achieves a similar result. Loop unrolling technique has been used to improve the execution time of the loops [14, 15, 20, 21, 22]. The number of unrolled iterations or the number of pipeline stages are both limited by the number of registers, by control dependencies (in the form of IF-THEN-ELSE), and by data dependencies. These issues will be further discussed in Section 3.2. But first we wish to compare the effectiveness of load pipelining and loop unrolling for the purpose of increasing the dependency distances in compiled codes.

3.1 Loop Unrolling Versus Load Pipelining

Loop unrolling and load pipelining are simple transformations which can be incorporated into a compiler. They differ slightly in the way they change the dependency distances in a loop. In Fig. 1, portions of the execution flow graph of a one-stage load-pipelined loop and of a one-time unrolled loop are compared with respect to the dependency distances they generate. In the case of

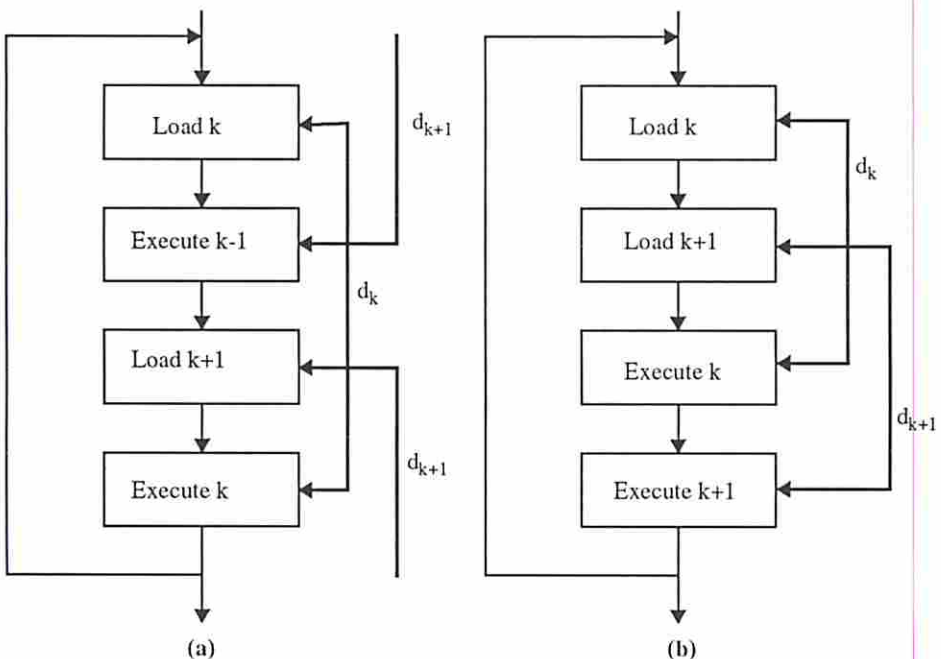

Fig. 1: Execution Flow for (a) Load Pipelining (b) Loop Unrolling where d_k represents a dependency distance of iteration k

load pipelining each additional stage increases the maximum dependency distance by an amount equal to the execution time of one whole iteration. On the other hand, loop unrolling reduces the execution time by deleting some of the branches that would normally have been executed in the original loop; but, as only half of the iteration body is overlapped with loads, the average dependency distance is half of what is achieved by load pipelining. For example, in Fig. 1 $d_k = d_{k+1}$ for load pipelining whereas $d_k < d_{k+1}$ for loop unrolling since the length of the 'load k+1' stage is usually much less than the length of the 'Execute k' stage. Another drawback of loop unrolling is the fact that dependency distances generated by loop unrolling are more variable because the loads of the original loop are not hoisted by the same amount in the unrolled loop. Fig 2 illustrates

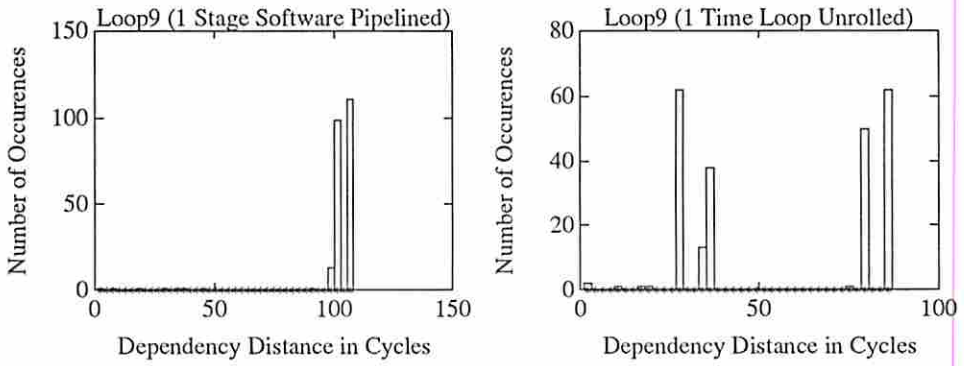

Fig. 2. Histograms of dependency distances of Loop 9 (for 1-stage pipelined and 1-time unrolled codes)

this problem and displays the histograms of dependency distances of Loop 9 after applying load pipelining and loop unrolling. Because of its advantages we have adopted load pipelining in our study.

3.2 Effects of Dependencies in the Code

In a program, control and data dependencies greatly affect the applicability of code transformations. The three types of data dependencies on memory locations are illustrated in Fig. 3. In an anti-dependency (Write After Read) data is first read and then modified by a following instruction; since the writes can be easily buffered in the processor, the write access in the dependency does not block the processor even if it misses in cache and therefore it does not need to be moved. For the same reason, output dependencies (Write After Write) do not cause any problem. True dependencies (Read After Write) are the most critical dependencies since the load instruction cannot be moved up beyond the preceding write access in the dependency. This is especially a prob-

. . . = X	X = . . .	X = . . .
X = . . .	X = . . .	. . . = X
(a)	(b)	(c)

Fig. 3. Data Dependencies (a) WAR (b) WAW (c) RAW

lem when the distance between the store and the load in the dependency is very short. In this paper we simply move loads which are at the end of a RAW dependency right up to the store (wherever possible) and do not attempt to move the store up.

A control dependency occurs when a conditional branch instruction tests the result of a

previous instruction. Conditional branch instructions[2] and their target instruction partition the program into *basic blocks*. It is usually very difficult to move a load outside of a basic block or across other basic blocks. However if a segment of the program contains an IF-THEN-ELSE block with forward jumps, we can move loads up across it and out of it. This situation is illustrated in Fig.4: we can move the load W instruction across the IF-THEN-ELSE block. We could

```
      .....
      load X(i)
      .....
      if (X(i)>1) goto L1
      load Z(k)
      goto L2
L1:   .....
      load Y(j)
L2:   .....
      load W(l)
```

Fig. 4: An IF-THEN-ELSE block with forward jump

move up the load Z or the load Y instructions or both. When loads are moved out of an IF-THEN-ELSE, we call them *speculative loads*, because we may be executing them uselessly, depending on the condition. In particular if we make both loads speculative one of them is useless at each execution of the IF-THEN-ELSE block and the new code generates more loads than the original code. In Fig.5, the IF-THEN-ELSE has a backward jump. In this case, we can also move the load

```
      . . . . . .
L1:   . . . .
      . . . . .
      load X(i)
      if (X(i)>1) goto L1
      load W(i)
```

Fig. 5: An IF-THEN-ELSE block with a backward jump

W instruction across the block if it does not have a dependency with another access in the block. Moving loads out of the block is more complex. If we move the 'load X' instruction above L1 it may not be executed or will be executed only once. Therefore, we can only move the load up to L1. Actually, loops can be generated by using IF-THEN-ELSE with backward jumps. If the compiler could detect loops generated by backwards goto's it could apply load pipelining. This has not been attempted for this paper.

Speculative loads can always be extracted from an IF-THEN-ELSE block with forward jumps. One reminder is that speculative loads may degrade performance and should be used only for high memory latencies. If the user can give hints to the compiler about the probabilities of executing the THEN or ELSE parts, the compiler may limit load hoisting to the part with higher probability.

---

2. Computed goto's should also been included in this discussion, but since there was none in the LL Loops we do not discuss them.

4 LAWRENCE LIVERMORE LOOPS

4.1 General Features of Lawrence Livermore Loops

Most of the Loops have only one nested DO-loop. Nearly all of the loops are generated by FORTRAN DO-loops. Only Loop 17 and outer loops of Loop 2 and Loop 16 use IF-THEN-ELSE blocks with backward jumps to generate loops. In Loops 13, 14 and 24 the indexes of array elements depend on variables that are calculated within the iteration body. Therefore, for these array elements, code transformations are hard to apply.

4.2 Classification of Lawrence Livermore Loops

After examining the features of the LL Loops in the context of non-blocking caches and applicability of code transformations, we have classified the LL Loops into 6 classes. In classes 1 to 4 the indexes of array variables are function of the loop indexes only. In class 5, the indexes depend on computed values. In class 6 there is no DO-loop, but rather a loop generated by IF-THEN-ELSE with backward jump. Because of the complexity associated with load hoisting for loops in class 5 (i.e., Loops 13, 14 and 24) and class 6 (Loop 17) we have not pursued them in this study. Distinctions between loop classes 1 to 4 are as follows:

1- Loops with no true dependency and no IF-THEN-ELSE block within iteration body.

2- Loops with no true dependency and at least one IF-THEN-ELSE block (which has no backward jump) within iteration body.

3- Loops with true dependencies either within or across iterations (No IF-THEN-ELSE block within iteration body)

4- Loops with both true dependencies and IF-THEN-ELSE blocks.

Table 2 shows the classification of the Loops. Note that only 4 Loops out of the 24 are considered too complex for our basic code motion strategy. Loop 2 and 16 have IF-THEN-ELSE with backward jumps but they also have regular DO-loops inside it, which can be. pipelined.

Table 1: Classification of Lawrence Livermore Loops

Loop No		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
Class No	1	x		x				x		x	x		x						x			x			
	2															x	x								
	3		x		x	x	x		x			x								x					
	4																				x		x	x	
	5													x	x										x
	6																	x							

4.3 Selected Lawrence Livermore Loops

From the above five classes the following five Loops are selected and used in our performance study (see Appendix for the codes of these Loops). From the first class Loop 1 and Loop 9 are selected. While Loop 1 has a small number of memory accesses within its iteration body, Loop 9 has a considerable amount of memory accesses. These two loops show the effect of register consumption, and the memory access stride. Four pipeline stages can be applied to Loop 1 whereas only one pipeline stage can be applied to Loop 9. From the second class, Loop 15 is selected. This Loop has a very large iteration body and uses a large number of integer and floating-point registers. From the third class we have selected Loop 11. This is a very small loop with true dependencies across iterations. This Loop shows the performance degradation due to true dependencies. Loop 20 is selected in the fourth class as its iteration size is larger than Loop 22 and as it has true dependencies both across and within iterations. Table 2 summarizes the characteristics of the chosen Loops (all entries except miss rates are per each iteration of the Loop before code transformation.) The average number of load misses is obtained for an 8 Kbyte direct-mapped cache with 32-byte blocks.

Table 2: Characteristics of selected loops (all numbers except miss rates are per iteration of the loop)

		Loop 1	Loop 9	Loop 11	Loop15	Loop 20
Register Consumption	Constant	12 Int., 3 Flt.	13 Int., 8 Flt.	7 Int., 1 Flt.	24 Int., 10 Flt.	18 Int., 1 Flt.
	Each Pipeline Stage	1 Int., 3 Flt.	1 Int., 10 Flt.	1 Int., 1 Flt.	1 Int., 11 Flt.	1 Int., 9 Flt.
Number of Load Accesses	No Dependence	3	10	1	6	7
	True Dependence			1		5
	Within IF-THEN-ELSE				5	1
Average Number of Load Misses		0.25	2.36	0.14	0.70	1.04
Execution Time in cycles (miss penalty =0)		38.02	101.55	15.83	495.48	123.14
Miss Rate		0.095	0.228	0.056	0.007	0.047

5 PERFORMANCE EVALUATION

5.1 General Definitions

We first define some of the terms used in the following sections. The *execution time* of a loop is the total number of processor cycles between the start of the execution till the end. This time includes both computation time and blocking time due to cache misses. The *speedup* is defined as the execution time on a system with a blocking cache divided by the execution time on a system with a non-blocking cache. The *miss overlap factor* is the average number of primary load misses pending whenever the processor is blocked.

5.2 Simulations

Trace-driven simulation is used in this study to evaluate the performance enhancement provided by code motion. Traces are obtained by running the selected Lawrence Livermore Loops on an instruction-level simulator of a SPARC processor with 32 registers and an 8kbyte direct-mapped cache with 32 byte blocks. The small data set of the LLLs forced us to choose a smaller than usual cache size in order to get meaningful results. To be able to make performance comparison of non-blocking caches with blocking caches we simulate two different systems. The system with blocking cache is shown in Fig. 6. The first level write-through cache and the second level

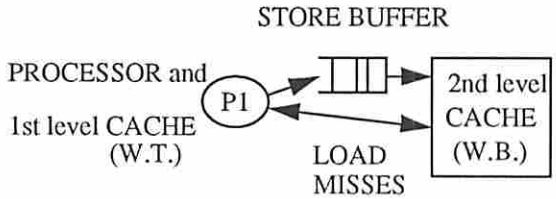

Fig. 6. Processor with caches and store buffer (W.T.: Write-Through; W.B.: Write-Back)

write-back cache have the same size, 8 KBytes, and are both direct-mapped. The processor has an infinite-size write buffer and therefore never blocks on a write. The only penalties come from load misses, which are blocking. This system can process loads that hit in the cache while there is a pending write miss. In the non-blocking cache system we have simulated the system described in Section 2 with one 8 Kbyte non-blocking direct-mapped cache; stores are buffered in an infinite-size store buffer so that their penalty is negligible, leaving only load miss penalties. Additionally, we assume an infinite number of interleaved memory banks (no bank conflicts). Even though a system with infinite number of interleaved memory banks is unrealistic, it helps us to focus on the effect of the software transformations to the dependency distances and the execution times.

The trace is made of two types of records. For each data access a record of the access is added to the trace and, for each load access, a record for the first following instruction using the value loaded is inserted in the trace.

In the SPARC simulator integer instructions are executed in one cycle, as well as load and store instructions provided they hit in cache. We assume that all instruction accesses hit in a separate instruction cache. This assumption removes the need to trace instructions and is justified by the high locality of instruction accesses in the Loops. Floating-point instructions may take 3 to 5 cycles and are not executed concurrently with integer instructions.

5.3 A Simple Model for Lockup-free Cache Performance

Without load hoisting, a lockup free cache is ineffective. When loads are hoisted and more pipeline stages are applied to the loop, the execution time is reduced and the speedup increases as shown in Fig. 7 for the case of Loop 1. For low memory latencies, the execution time curve is almost flat and the speedup increases roughly linearly; the reason is that the memory latency is less than the dependency distance of most loads. For memory latencies larger than the maximum dependency distance, the program still benefits from load hoisting but the execution time increases linearly with the latency. Loop 1 has no feature hampering load hoisting and therefore

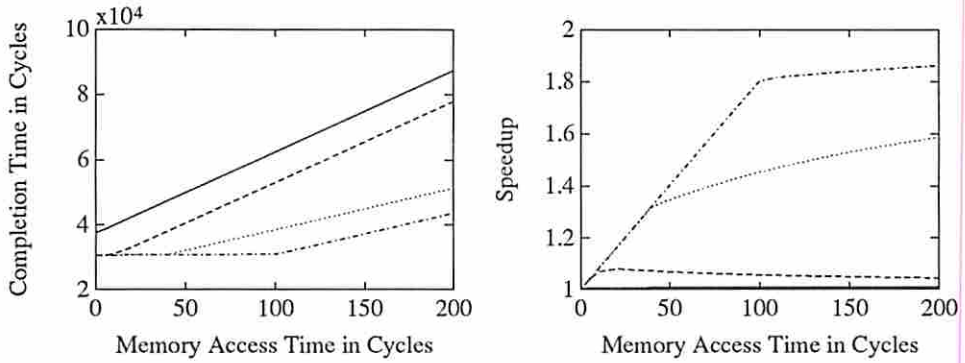

Fig.7. Effect of load hoisting and pipelining on Loop 1
Solid line: No load hoisting. Dashed line: Load hoisting, 0 pipeline stage.
Dotted line: 1 pipeline stage. Dashdot line: 3 pipeline stages.

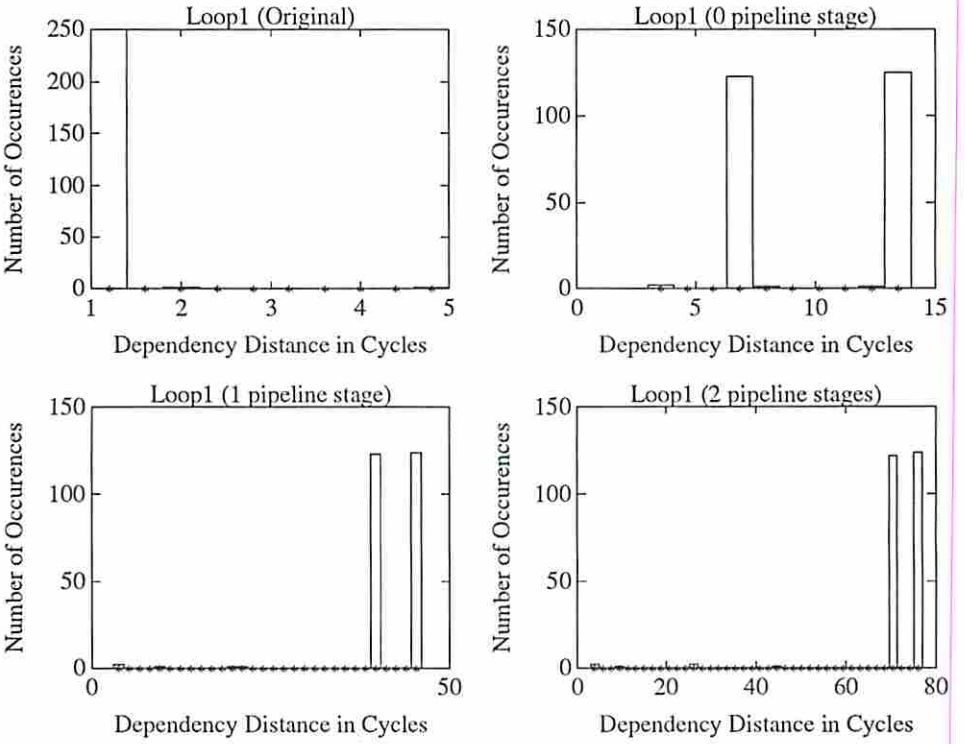

Fig. 8. Histogram of dependency distances of Loop 1 for various degrees of load hoisting/pipelining

all loads can be moved by the same amount. The histogram of dependency distances for Loop 1 is shown in Fig. 8 for various degrees of load pipelining. As can be seen, the dependency distances are clustered around one value, which increases with the number of pipeline stages. The break-point in Fig. 7 occurs for a value of memory latencies equal to the center of this cluster. In general, however, besides loads that can be moved without restrictions, a Loop contains loads that cannot be moved or that can be moved by a limited amount. This is reflected in the histograms of dependency distances of other Loops (Fig. 14, 16, 17, and 18).

The histograms show that the dependency distances are grouped in a few clusters. Based on this observation, the curves for the execution time and the speed up can be explained by the

following simple model.

Let T be the miss latency and T_d be the average dependency distance. We distinguish between $T \leq T_{min}$, $T_{min} < T \leq T_{max}$, and $T > T_{max}$, where T_{min} and T_{max} correspond to the minimum and maximum dependency distances.

Case 1: $T \leq T_{min}$

In this case, the dependency distances are long enough that all load misses are overlapped with execution. If T_{ex} is the execution time of the program when all loads hit in the cache and if M is the total number of load misses, the execution times are $T_{ex} + M \times T$ and T_{ex}, for the blocking and the non-blocking caches, respectively. Therefore the speedup is:

$$Sp = \frac{T_{ex} + M \times T}{T_{ex}} = 1 + M \times T^0 \quad \text{with} \quad T^0 = T/T_{ex}$$

Case 2: $T_{min} < T \leq T_{max}$

Let's assume that there are L clusters of dependency distances in the code where each cluster j contains R_j number of elements, so that the total number of load misses M is $M = \sum_{j=1} R_j$

In this case the load misses with dependency distances larger than T are covered by execution and do not block the processor. Assuming that there is no overlap among the loads the load misses with dependency distances less than T block the processor for a period of $(T - T_j)$ where T_j is the average dependency distance of the cluster j. However, because of the overlap among load misses, the actual blocking time due to loads in cluster j is $\alpha_j R_j (T - T_j)$ where α_j is less than one. There may be more than one cluster with dependency distance less than T. Therefore, the blocking time due to all of these clusters is $\sum_{j=1}^{i} \alpha_j R_j (T - T_j)$

For each memory latency T, i is the number of clusters with dependency distances less than T. Then the speedup for this region becomes

$$Sp = \frac{T_{exe} + M \times T}{T_{exe} + \Delta T_{exe} + \sum_{j=1}^{i} \alpha_j R_j (T - T_j)} = \frac{1 + M \times T^0}{1 + \Delta T^0_{exe} + \sum_{j=1}^{i} \alpha_j R_j (T^0 - T^0_j)}$$

where, $T^0_j = T_j / T_{exe}$, and $0 < \alpha_j \leq 1$. ΔT_{exe} is the increase in the execution time due to the execution of speculative loads. $\Delta T^0_{exe} = \Delta T_{exe} / T_{exe}$ is the normalized execution overhead resulting from speculative loads.

Case 3: $T > T_{max}$

In this case, all dependency distances are too short to cover the miss latency and the average penalty of the non-blocking cache is $T - T_d$ if we assume that one single load miss can be pending at any one time. However, this penalty is further reduced because of the overlap among multiple

load misses. We denote the miss overlap factor by f. The average penalty of a load miss in the lockup-free cache is $(T - T_d)/f$ and the execution time is $T_{ex} + M \times (T - T_d)/f$; therefore, the speedup is

$$Sp = \frac{T_{ex} + M \times T}{T_{ex} + \Delta T_{exe} + M \times (T - T_d)/f} = \frac{1 + M \times T^0}{1 + \Delta T^0_{exe} + M \times (T^0 - T^0_d)/f}$$

As T becomes very large (with respect to T_d) the speedup goes to f, the miss overlap factor.

In the case of Loop 1 (Fig. 8) and Loop 9 (Fig. 16), which both have a single cluster of dependency distances, cases 1 and 3 above apply. The latency equal to T_d clearly divides the plots of Fig. 7 and Fig. 10 into two regions corresponding to cases 1 and 3 above.We call it the *critical latency*. In Loop 11, there are two clusters (the one at 0 corresponds to the load with a RAW dependency) in the histogram (see Fig. 14) and the division of the speedup plots in the two regions is visible if not as sharp as in Fig. 11. The same applies to all execution time and speedup curves for the other Loops. In general, a critical latency can be observed beyond which the shape of the speedup curve is dictated by the miss overlap factor.

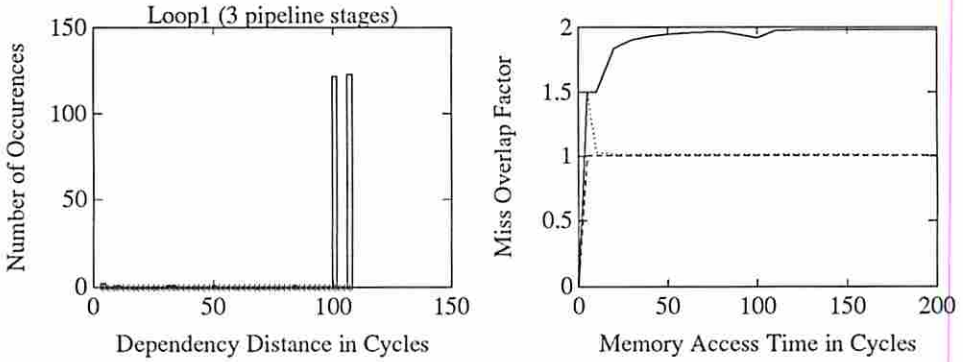

Fig. 9. (a) Histogram of dependency distances of Loop 1 (3 stage software pipelined)
(b) Miss overlap Factor for Loop 1. Dashed line: Original Code.
Dotted Line. Load hoisting. Solid line: 3 stage pipelined

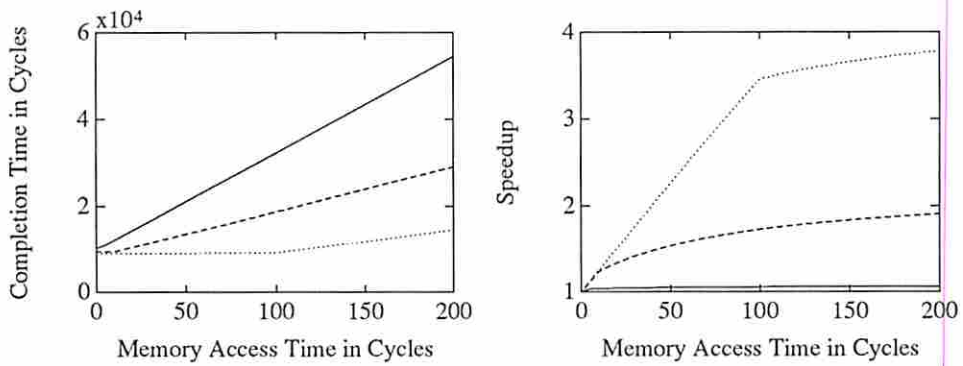

Fig. 10. Effect of load hoisting and pipelining on Loop 9
Solid line: No load hoisting. Dashed line: Load hoisting, 0 pipeline stage.
Dotted line: 1 pipeline stage.

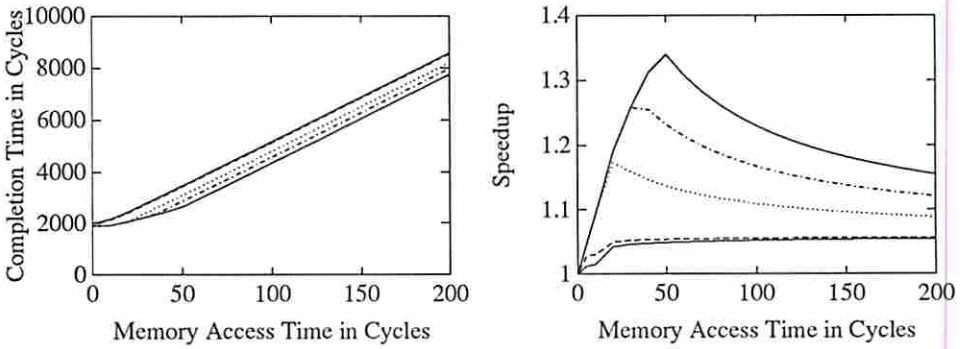

Fig.11. Effect of load hoisting and pipelining on Loop 11
First Solid line: No load hoisting. Dashed line: Load hoisting, 0 pipeline stage.
Dotted line: 1 pipeline stage. Dashdot line: 2 pipeline stages. Last Solid line: 3 pipeline stages.

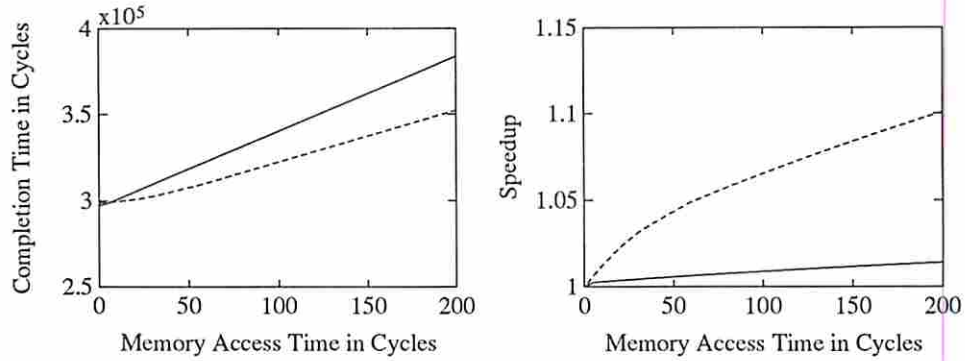

Fig. 12. Effect of load hoisting and pipelining on Loop 15
Solid line: No load hoisting. Dashed line: Load hoisting, 0 pipeline stage.

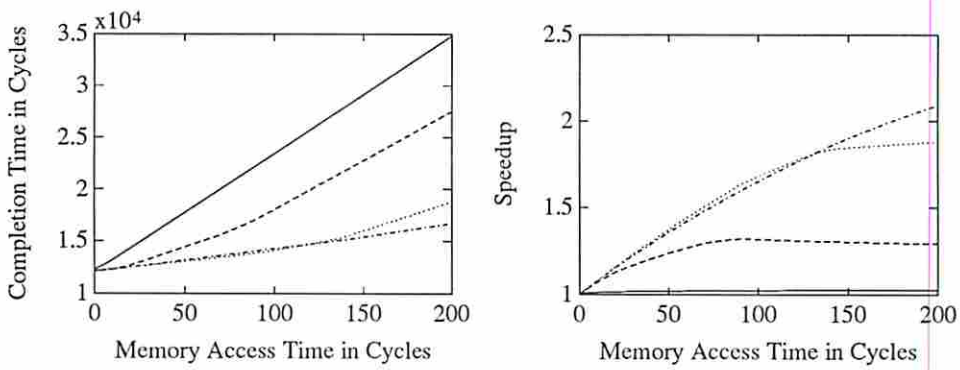

Fig. 13. Effect of load hoisting and pipelining on the Loop 20
Solid line: No load hoisting. Dashed line: Load hoisting, 0 pipeline stage.
Dotted line: 1 pipeline stage. Dash-dot line: 2 pipeline stages

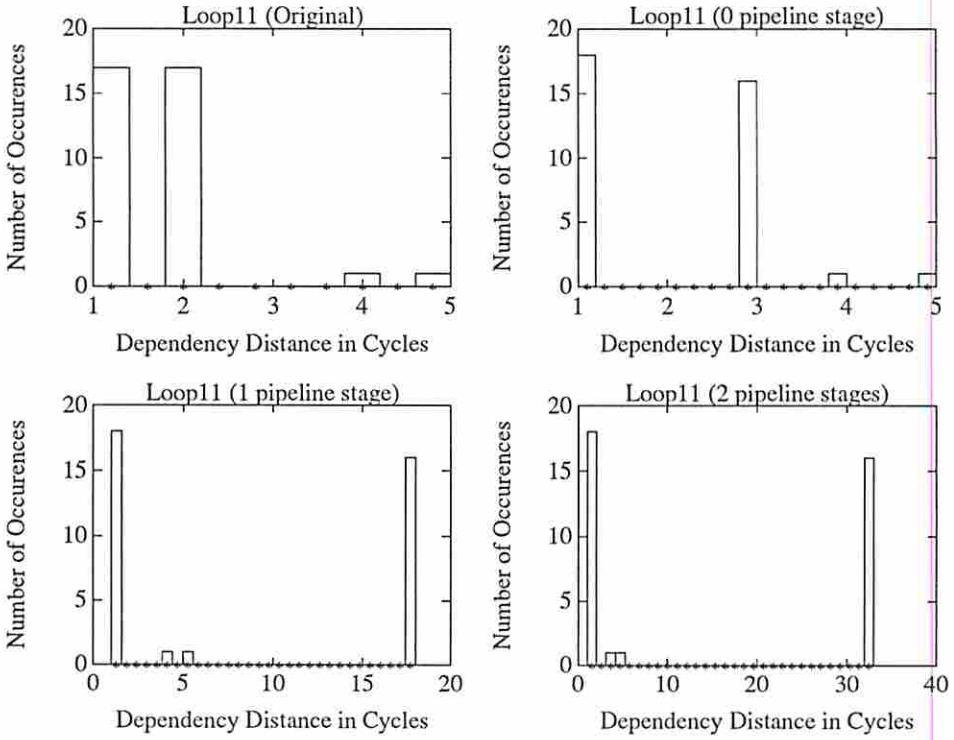

Fig. 14. Histogram of dependency distance of Loop 11 for various degrees of load hoisting/pipelining.

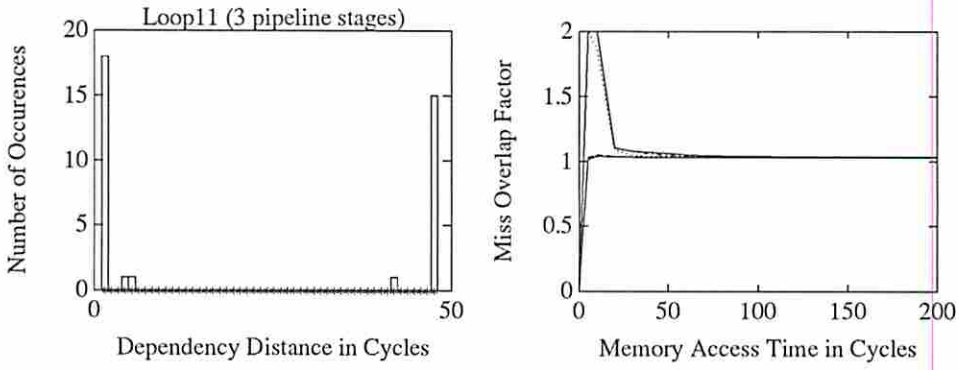

Fig. 15. (a) Histogram of dependency distances of Loop 11 (3 stage software pipelined)
(b) Miss overlap Factor for all codes of Loop 11

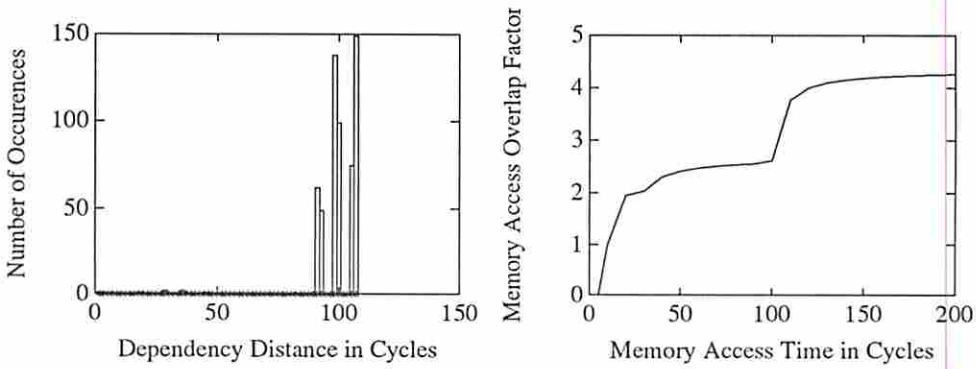

Fig. 16. Histogram of dependency distances and miss overlap factor for Loop 9
(1 pipeline stage)

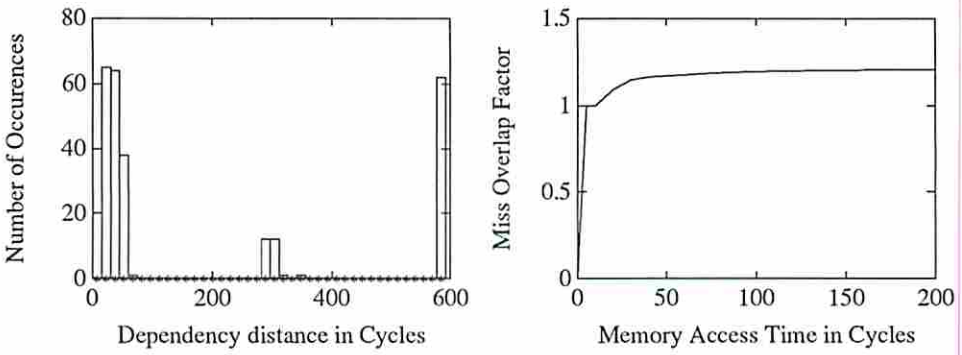

Fig. 17. Histogram of dependency distances and miss overlap factor for Loop 15 (0 pipeline stage)

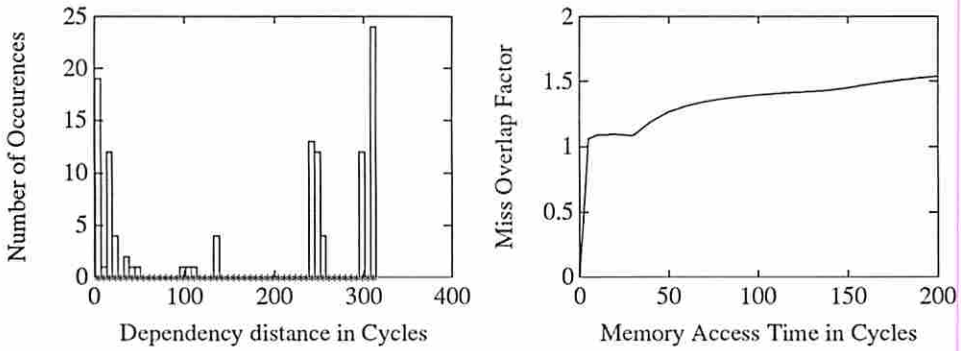

Fig. 18. Histogram of dependency distances and miss overlap factor for Loop 20 (2 pipeline stages)

5.4 Discussion of Simulation Results

The results discussed in this section are obtained by manually applying code motion to the optimized assembly code generated by the compiler for the five selected Loops. All loads are moved up in the code, irrespective of whether they hit or not. When a store instruction blocks the motion of a load (RAW dependency), we stop the movement of that load just after the store and do not attempt to move the store. When moving a load its address calculation is also moved up. Finally, in order to be able to pipeline the codes further we changed all double precision floating-point variables and operations to single precision.

The execution time and speedup curves of all selected Loops (see Figs. 7, 10, 11,12, and 13) exhibit two different behaviors for low and high memory access latencies. For low latencies the speedup increases linearly with increasing latency and the execution time remains nearly constant. For high latencies the speedup curve flattens and reaches a constant value and the execution time starts increasing linearly.As demonstrated by the model the gain in the low latency region is mainly due to the overlap of loads withexecution.In contrast, the gain in the high latency region is due to the overlap of multiple loads and the speedup tends to the average miss overlap factor. In all cases, a critical latency separating the two regions can be identified in the graphs. The critical latencies and miss overlap factors for all cases are shown in Table 3.

Among all examined Loops, Loop 1 and Loop 9, which are in the same class, exhibit the best behavior. In these Loops each additional pipeline stage increases the dependency distances by a constant amount of cycles as a whole iteration body is inserted with each additional pipeline stage between the load of a value and the instruction which uses that value For Loop9 the number of registers prevents us from pipelining more than one stage.The miss overlap factor,.which is

Table 3: Critical latencies and miss overlap factor for various degrees of load hoisting/pipelining

Loop No	Critical Latency (cycles)					Miss Overlap Factor at 200 cycles latency				
	Original	Load Hoisting	1 Stage Pipeline	2 Stage Pipeline	3 Stage Pipeline	Original	Load Hoisting	1 Stage Pipeline	2 Stage Pipeline	3 Stage Pipeline
1	1	5	30	70	100	1.004	1.012	1.984	1.985	1.983
9	1	10	100			1.067	2.242	4.292		
11	2	2	20	30	48	1.030	1.030	1.031	1.032	1.034
15	5	20				1.017	1.208			
20	2	14	131	244		1.0176	1.246	1.538	1.538	

important at very large latencies, increases with the degree of pipelining. However this increase is gradual and slow. This can be explained from Table 2 by the average number of misses in each iteration of the loop. It takes four iterations of Loop 1 per additional miss, while Loop 9 suffers two misses at each iteration. We can therefore expect that the miss overlap factor increases by one for every four additional pipeline stages in Loop 1 and that it increases by two for each additional pipeline stage of Loop 9.

Loop 11 gains very little from pipelining the loads. One reason is the short execution time of the Loop iteration, which provides only 15 cycles of overlap per pipeline stage; Loop 11 uses few registers and more pipelining would be possible and useful. Another reason is the RAW dependency which limits load hoisting to one of the two loads. It will take about nine stages of load pipelining to raise the miss overlap factor to two.

Loop 15 is in class 2 and has IF-THEN-ELSE blocks with forward jumps. This loop has a large loop body, and it uses nearly half of the floating-point registers per iteration, which prevented us from pipelining; loads are simply hoisted in the loop body. For this reason the critical latency is limited to 20 cycles, which is very small. From the miss overlap factor we can also state that the speedup will reach 1.25 for very large latencies. Many loads are hoisted out of the IF-THEN-ELSE blocks and become speculative loads. The curves show that these extra loads do not degrade performance in the case of this Loop. The only way to obtain better results for Loop 15 is to utilize registers better and only pipeline loads that miss.

The last loop that we have examined is from class 4. Loop 20 has true dependencies across and within iterations. Within its body Loop 20 also has an IF-THEN-ELSE with forward branches and there is an array access in the THEN part. In order to compare the performance of the codes with and without speculative loads we have developed two versions of Loop 20.The first version has no speculative loads; all loads which are not in an IF-THEN-ELSE block are moved and pipe-

lined. The speedup and execution time curves show that even a one-stage pipeline has a large critical latency of 150 cycles (and of 244 cycles in the case of a two-stage pipeline). For very high latencies the speedups of both one-stage and two-stage pipelined loops tends to two. In the second version of the code we have hoisted the loads in the THEN part of the IF-THEN-ELSE block. The curves show that the performance of the two-stage pipelined code with speculative loads is better than the one with no speculative loads for latencies higher than 20 cycles (see Fig. 19). For very low latencies the overhead of these extra loads (i.e., execution of extra instructions for address calculation and of the load itself) degrades the performance of the loop.

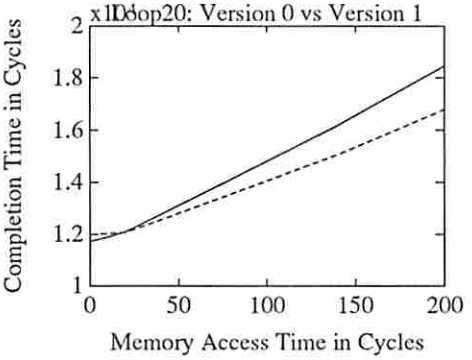

Fig. 19. Effect of speculative loads on Loop 20 (2 stage pipelined)
Solid Line: No speculative load Dashed line: One speculative load

6 POSSIBLE IMPROVEMENTS

In this Section we propose solutions to the problems we met in applying a simple code motion transformation to the FORTRAN LL Loops. The first problem was the limit set by the number of registers. Each load pipeline stage consumes a number of registers equal to the number of load instructions in the iteration body (assuming single precision operations). An obvious hardware solution is to increase the number of registers, which requires to change instruction formats. A second solution is to find compiler methods which reduce register consumption. For example, presently, all loads are moved up in the code whether they hit or miss. The compiler should first detect the loads which are most likely to miss (Porterfield's Overflow Iteration in [10] is such a technique; simpler approaches may also be very effective) and apply load pipelining to these loads only. As there will always be a limit on the number of registers we may not be able to create enough pipeline stages for very high latencies. In this case, the execution time becomes negligible compared to the miss latency and the compiler should issue in-cache loads instead of in-register loads.

The second problem we encountered is the true dependencies in the code. The problem caused by RAW dependencies can be solved in hardware by *write posting*. With write posting the cache controller allocates a block on a write miss, stores the value in the block, marks the modified word as full and sends a block request to memory. Any word marked full in a pending block can be read. However a read access to an empty word of a pending block causes a secondary miss. Write posting makes the cache more complex but is very effective for RAW dependencies. A method requiring only the compiler support is keeping the value in the register which is the source of the store until the instruction that reads the value is executed (this effectively eliminates the

read in the RAW dependency); if the load is very far from the store the miss started by the store may be completed by the time the load is issued. Removing the load and keeping the value in the register live is effective for short loops since the load is removed, which saves its execution plus its address computation; for longer loops, reserving one extra register for this purpose may prevent some useful compiler transformations and therefore it may be better to release the register and hoist the load. The trade off must be determined by the compiler. Finally, the last and the most difficult compiler solution is to move up the store instruction along with the load instruction and/or to push the instructions which use the loaded data value down in the code.

The third problem is the control dependencies due to IF-THEN-ELSE. Loads moved out of IF-THEN-ELSE blocks are speculative. We have not moved loads out of IF-THEN-ELSE blocks with backward jumps. At this point this problem seems to be the most challenging one. Note however that only Loop 17 makes exclusive use of such constructs to implement DO-loops.

The fourth problem concerns loads whose address cannot be known at compile time because it is a function of the variables computed in the loop. Loops 13, 14, and 24 have such loads. Pipelining these loads would require to apply software pipelining at the source code level first before the code hoisting procedure.

As a final point consider the limitations of the load pipelining technique. We may not be able to apply it directly if the loop size is very large; or its effect may not be sufficient in the case of a loop with a short body. In these cases, other code transformation techniques at the source code level can be combined with load pipelining. For example, if the loop body is too large, loop distribution can be applied to the loop to divide one big loop into two smaller loops. If the loop size is too small, loop fusion, in which two small loops with the same index values are combined to generate a bigger loop, can be first applied to two successive loops. Loop fusion also increases the total number of pending memory accesses which in turn increases the speedup in system with a high miss latency.

7 CONCLUSION

Lockup-free caches aim to solve the speed gap problem of current (and future) computer systems. In this paper we proposed a combined non-blocking cache/processor architecture which eliminates the complexity of lockup-free caches. To be effective, lockup-free caches need substantial compiler support. Current compilers generate code for systems with blocking loads. Therefore, generated codes have very small dependency distances. To increase the dependency distances compilers for non-blocking cache/processor architectures must apply some program transformations, such as load pipelining and hoisting. The performance improvements that can be obtained by means of code transformations and non-blocking loads are limited by the total number of floating-point and integer registers. As it is difficult to increase the number of registers of a system, the compiler transformation must use registers sparingly.

More work is needed to refine the cache architecture and compiler transformations. However, the performance results introduced in this paper show that lockup-free caches can be very effective in systems with large miss latencies.

8 SELECTED REFERENCES

[1] W. Brantley, K. McAuliffe, and T. Ngo. *RP3 processor-memory element.* Proc. Int. Conf. on Parallel Processing, pages 782-789, April 1985.

[2] A. Chang and M. Mergen. *801 Storage: Architecture and Programming.* ACM Transactions on Computer Systems, 6(1):28-50, February 1988.

[3] W. Connors, J. Florkowksi, and S. Patton. *The IBM3033: An Inside look.* Datamation, pp. 198-218, May 1979.

[4] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W.D. Weber. *Comparative evaluation of latency reducing and tolerating techniques.* 18th International Symposium on Computer Architecture, May 1991.

[5] M. Franklin and G. Sohi. *Non-blocking caches for high performance processors*, 1991. Manuscript.

[6] A. C. Klaiber and H. M. Levy. *An architecture for software-controlled data prefetching. 18th International Symposium on Computer Architecture*, May 1991.

[7] Kogge, P. M. *The Architecture of Pipelined Computers.* McGraw-Hill, 1981.

[8] Kroft, D. *Lockup-free Instruction Fetch/Prefetch Cache Organization*, Proc. of the 8th Ann. Int. Symp. on Comp. Arch., pp. 81-87, June 1981.

[9] McMahon, F. H. *LLNL Fortran Kernels: MFlops.* Technical Report, Lawrence Livermore Laboratories, Livermore, CA, March 1984.

[10] Porterfield, A. K. *Software Methods for Improvement of Cache Performance on Supercomputer Applications.* PhD dissertation, RICE COMP TR 89-93, May 1989.

[11] Scheurich, C. and Dubois, M. *Lockup-free Caches in High-Performance Multiprocessors*, J. of Par. and Dist. Comp., Jan. 1991.

[12] H. Warren Jr. *Instruction scheduling for the IBM RISC System/6000 processor.* IBM J RES DEVELOP, 34(1), pp. 85-91, January 1990

[13] Jean-Loup Baer, Tien-Fu Chen. *An Effective On-Chip Preloading Scheme To Reduce Data Access Penalty,* Proc. 1990 International Conference on Supercomputing, pp. 354-368, 1990.

[14] A. E. Charlesworth. *An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS-164 Family,* Computer, Vol. 14, No. 9, September 1981.

[15] Shlomo Weiss and James E. Smith. *A Study of Scalar Compilation Techniques For Pipelined Supercomputers,* Int. Conf. on Architectural Support for Programming Languages and Operating Systems, October 1987.

[16] R. A. Mueller, M. R. Duda, P. H. Sweany, and J. S. Walicki. *Horizon: A Retargetable Compiler for Horizontal Microarchitectures,* IEEE Transactions on Software Engineering, Vol. 14, No. 5, May 1988.

[17] Robert A. Mueller and Dan Budge. *Automated Optimization Refines SIMD Microcode For Efficient Programming,* Computer Design, Vol. 27, No. 18, pp. 67-72, October 1988.

[18] M. Lam. *Software Pipelining: An Effective Scheduling Technique for VLIW Machines,* ACM SIGPLAN Notices, 23(7), pp. 318-328, July 1988.

[19] R. F. Touzeau. *A FORTRAN Compiler For The FPS-164 Scientific Computer*, Proc. of the ACM SIGPLAN 84 Symposium on Compiler Construction, June 1984.

[20] J. A. Fisher, *Very Long Instruction Word Architectures and the ELI-512,* 10th Annual International Symposium on Computer Architecture, Stockholm, Sweden, June 1983.

[21] J. J. Dongarra and A.R. Hinds, *Unrolling Loops in FORTRAN,* Software-Practice and Experience, Vol. 9, No. 3, March 1979.

[22] R. L. Lee, A. Y. Kwok, and F. A. Briggs. *The Floating Point Performance of a Superscalar SPARC Processor,* 18th International Symposium on Computer Architecture, pp. 28-37, May 1991.

[23] A. Agarwal, et al. *The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor,* Scalable Shared Memory Multiprocessors, pp. 239-261, Kluwer Academic Publishers, 1991.

[24] J. E. Smith, *Decoupled Access/Execute Computer Architectures,* ACM Transactions on Computer Systems, Vol. 2, No. 4, pp. 289-308, November 1984.

[25] J. R. Goodman, et al. *PIPE: A VLSI Decoupled Architecture,* International Symposium on Computer Architecture, pp. 20-27, 1985.

9 APPENDIX

The following are the codes of the selected LLLs:

```
      subroutine Loop1(990,Q,R,T,X,Y,Z)
      integer n,k
      double precision Q,R,T,X(1001),Y(1001),Z(1001)
      do 1 k = 1,990
1         X(k) = Q + (Y(k) * ((R * Z(k+10)) + (T * Z(k+11))))
      return
      end
```

Fig. A1. Fortran code for Loop 1

```
      subroutine Loop9(101,CO,DM22,DM23,DM24,DM25,DM26,DM27,DM28,PX)
      integer i,n
      double precision CO,DM22,DM23,DM24,DM25,DM26,DM27,DM28,PX(25,101)
      do 9 i = 1,101
         PX(1,i) = PX(3,i) + (CO * (PX(5,i) + PX(6,i))) +
     .      (DM28 * PX(13,i)) + (DM27 * PX(12,i)) +
     .      (DM26 * PX(11,i)) + (DM25 * PX(10,i)) +
     .      (DM24 * PX(9,i)) + (DM23 * PX(8,i)) +
     .      (DM22 * PX(7,i))
9     continue
      return
```

Fig. A2. Fortran code for Loop 9

```
      subroutine Loop11(128,X,Y)
      integer k,n
      double precision X(1001),Y(1001)
      X(1) = Y(1)
      do 11 k = 2,n
11        X(k) = X(k-1) + Y(k)
      return
      end
```

Fig. A3. Fortran code for Loop 11

```
      subroutine Loop20(100,DK,S,T,G,U,V,VX,W,X,XX,Y,Z)
      integer k,n
      double precision DK,DI,DN,S,T,G(1001),U(1001),V(1001),
     .     VX(1001),W(1001),X(1001),XX(1001),Y(1001),Z(1001)
      do 20 k = 1,n
         DI = Y(k) - (G(k) / (XX(k) + DK))
         DN = 0.2d0
         if (DI .NE. 0.0d0) DN = max(S, min(Z(k)/DI, T))
         X(k) = ((W(k) + V(k) * DN) * XX(k) + U(k)) /VX(k) + V(k) * DN)
         XX(k+1) = (X(k) - XX(k)) * DN + XX(k)
20    continue
      return
      end
```

Fig. A4. Fortran code for Loop 20

```
      subroutine Loop15(101,VF,VG,VH,VS,VY)
      integer j,k
      double precision R,S,T,VF(101,7),VG(101,7),VH(101,7),VS(101,7),VY(101,25)
      do 1500 j = 2,7
      do 1500 k = 2,n
          if (j-7) 1502,1501,1501
1501      VY(k,j) = 0.0d0
          goto 1500
1502      if (VH(k,j+1) - VH(k,j)) 1504,1504,1503
1503      T = 0.053d0
          goto 1505
1504      T = 0.073d0
1505      if (VF(k,j) - VF(k-1,j)) 1506,1507,1507
1506      R = max(VH(k-1,j), VH(k-1,j+1))
          S = VF(k-1,j)
          goto 1508
1507      R = max(VH(k,j), VH(k,j+1))
          S = VF(k,j)
1508      VY(k,j) = dsqrt(VG(k,j)**2 + R * R) * T / S
1509      if (k-n) 1511,1510,1510
1510      VS(k,j) = 0.0d0
          goto 1500
1511      if (VF(k,j) - VF(k,j-1)) 1512,1513,1513
1512      R = max(VG(k,j-1), VG(k+1,j-1))
          S = VF(k,j-1)
          T = 0.073d0
          goto 1514
1513      R = max(VG(k,j), VG(k+1,j))
          S = VF(k,j)
          T = 0.053d0
1514      VS(k,j) = dsqrt(VH(k,j)**2 + R * R) * T / S
1500  continue
      return
      end
```

Fig. A5. Fortran code for Loop 15