# Hiding Memory Latency using Dynamic Scheduling in Shared-Memory Multiprocessors

Kourosh Gharachorloo, Anoop Gupta, and John Hennessy

Computer Systems Laboratory
Stanford University, CA 94305

## Abstract

The large latency of memory accesses is a major impediment to achieving high performance in large scale shared-memory multiprocessors. Relaxing the memory consistency model is an attractive technique for hiding this latency by allowing the overlap of memory accesses with other computation and memory accesses. Previous studies on relaxed models have shown that the latency of write accesses can be hidden by buffering writes and allowing reads to bypass pending writes. Hiding the latency of reads by exploiting the overlap allowed by relaxed models is inherently more difficult, however, simply because the processor depends on the return value for its future computation.

This paper explores the use of dynamically scheduled processors to exploit the overlap allowed by relaxed models for hiding the latency of reads. Our results are based on detailed simulation studies of several parallel applications. The results show that a substantial fraction of the read latency can be hidden using this technique. However, the major improvements in performance are achieved only at large instruction window sizes.

## 1 Introduction

The performance of large scale shared-memory multiprocessors can be severely limited by the long latency of remote memory accesses. Overlapping of memory accesses with other computation and accesses is an attractive method for hiding the memory latency. The degree to which such overlap is allowed within the same thread is determined by the memory consistency model chosen for the architecture.

The *sequential consistency* (SC) model [22] proposed by Lamport is the most commonly assumed memory consistency model; it requires the execution of a parallel program to appear as some interleaving of the execution of the parallel threads on a sequential machine. Unfortunately, this requirement places severe restrictions on the allowable ordering of shared memory accesses within the same thread. To allow for more overlap among memory accesses, several relaxed consistency models have been proposed [1, 6, 10, 12]. One of the more relaxed models is *release consistency* (RC) [10]. RC allows for significant overlap among memory accesses as long as synchronization operations are appropriately identified.

Relaxed models such as release consistency allow read and write accesses between synchronizations to be overlapped with one another, thus presenting the opportunity to hide the latency arising from both read and write accesses. Whether this opportunity is exploited depends on the aggressiveness of the processor architecture and the memory subsystem. For example, a processor that blocks for the return value of a read access (typical for current commercial processors) cannot hide the latency of the read. Similarly, a cache that stalls while servicing a cache miss cannot exploit the allowable overlap among multiple misses.

The performance gain from relaxing the consistency model has been previously studied for processors that block on read accesses [7]. These studies have shown that the latency of writes can be effectively hidden as long as the write buffer and the cache allow reads to be serviced while there are pending writes. However, exploiting relaxed models for hiding the latency of reads remains unexplored.

Hiding the latency of reads through overlap with other computation and accesses is inherently more difficult than hiding write latency, simply because the computation following the read usually depends on the return value. To hide this latency, the processor needs to find other independent computation and memory accesses to process while awaiting the return value for the read. To achieve this in hardware implies an aggressive processor architecture with a decoupled decode and execution unit and capability for dynamic scheduling, branch prediction, and speculative execution. Given expected memory latencies ranging from tens to hundreds of processor cycles, there are serious concerns as to whether such a processor architecture allows for effective overlap of read latencies and whether the added complexity is justifiable.

This paper explores the use of dynamically scheduled processors to exploit the overlap allowed by relaxed models for hiding the latency of reads. The study is based on detailed simulation results of five parallel applications. The applications are a particle-based simulator used in aeronautics (MP3D) [26], an LU-decomposition program (LU), a digital logic simulation program (PTHOR) [35], a standard cell placement and routing program (LOCUS) [30], and a program for simulating large scale ocean currents (OCEAN) [31]. The simulated processor environment is based on an architecture proposed by Johnson [18].

Our results show that dynamic scheduling in combination with relaxed consistency models is successful in hiding a substantial fraction of the read latency. However, to fully realize these gains requires large instruction lookahead windows. The window size is varied from 16 to 256 in our experiments. With memory latency of 50 cycles, we observe that there is a noticeable gain from increas-



© 1992 ACM 0-89791-509-7/92/0005/0022 $1.50

ing the window size from 16 to 32 and from 32 to 64 instructions. However, the performance levels off past a window size of 64. Our results for a latency of 100 cycles show similar trends [9]. The lower performance at the smaller window sizes is primarily attributed to data dependences at short distances and the inability of the processor to overlap independent operations that are farther apart than the size of the lookahead window. Except for PTHOR, the branch behavior of the applications does not hinder lookahead until we reach very large window sizes.

The next section provides background information on memory consistency models and describes architectural features that are necessary to exploit the full potential of a relaxed consistency model. Section 3 presents details of the processor architecture used in our study. The simulation environment and the five parallel applications are also described in this section. Simulation results are presented in Section 4. Sections 5 and 6 discuss the results and the related work. Section 7 concludes the paper.

# 2 Background

This section begins with a brief description of several memory consistency models that have been proposed in the literature. The latter part of the section discusses processor and memory system features that are necessary to fully exploit the overlap of accesses and computation allowed by the relaxed consistency models.

## 2.1 Memory Consistency Models

Overlapping memory accesses is an attractive technique for hiding the large memory latency in multiprocessors. However, as a result of the distribution of memory and interconnection resources, multiple requests issued by a processor may complete out of order. This can lead to incorrect behavior if a program depends on completion of accesses in a certain order. Consequently, we need to impose certain restrictions on the completion order of memory accesses to provide a reasonable programming model for an architecture. This function is fulfilled by the memory consistency model.

A memory consistency model specifies the constraints on ordering of shared memory accesses. *Sequential consistency* (SC) [22] is the most commonly assumed memory model. Sequential consistency requires the execution of a parallel program to appear as some interleaving of the execution of the parallel processes on a sequential machine. *Processor consistency* (PC) [12] relaxes some of the restrictions of SC. While PC requires writes issued from a process to be seen in program order by all other processes, it allows reads to be reordered with respect to previous writes from a process. The constraints for satisfying PC are specified formally in [10].

Dubois et al. [6] have proposed a further relaxation of consistency constraints by relating ordering to synchronization points in the program. The *weak ordering* (WO) model (also known as weak consistency) is based on this idea and ensures consistency only at synchronization points. WO provides considerable flexibility for overlapping accesses between synchronization points. The *release consistency* model (RC) [10] extends this framework by further classifying synchronization into *acquires* and *releases*. An acquire is a read operation that gains permission to access a set of data (e.g., a lock operation or waiting for a flag to be set), while a release is a write operation that gives away such permission (e.g., an unlock or setting of a flag). This information is used to further relax the ordering between data and synchronization accesses. The *data-race-free-0* (DRF0) [1] model proposed by Adve and Hill is similar to RC, although it does not explicitly distinguish between acquire and release accesses.

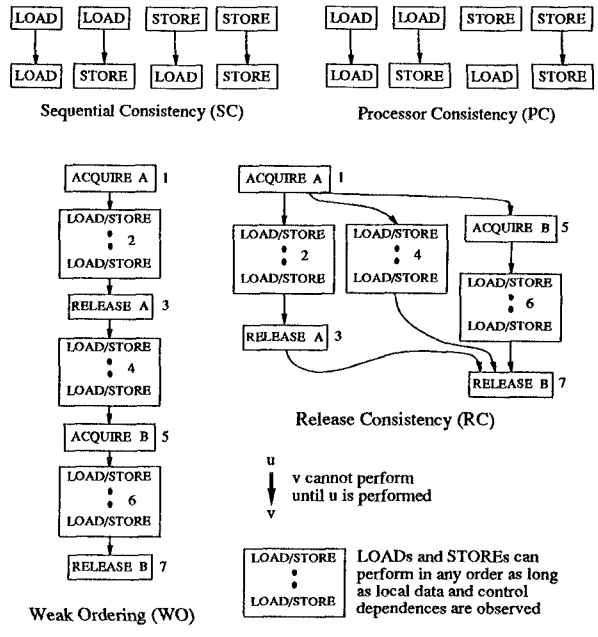


Figure 1: Ordering restrictions on memory accesses. The numbers beside the blocks indicate the program order.

The constraints imposed by a consistency model can be satisfied by restricting when an access is allowed to perform. A read is considered *performed* (or completed) when the return value is bound and cannot be modified by other write operations while a write is considered *performed* (or completed) once the value written is visible to all processors. Figure 1 shows the restriction on completion order of accesses from the same process for straightforward implementations of the four consistency models discussed above.

As shown in Figure 1, SC can be guaranteed by performing accesses within each process in a serial manner. PC provides more flexibility by allowing reads to bypass writes in the same process. WO and RC provide further flexibility by exploiting information about synchronization accesses. Both models allow accesses between synchronization to be overlapped. The figure also shows how RC exploits the acquire and release information to further relax ordering constraints.

While relaxed models such as WO and RC permit considerable overlap among shared accesses, whether such overlap takes place depends on the aggressiveness of the processor architecture and the memory system. The next section describes the corresponding design issues.

## 2.2 Processor Architecture and Memory System Issues

To effectively hide the latency of memory accesses, the processor requires the ability to continue past pending accesses to find other computation and memory accesses to execute. The extent of this ability is primarily determined by the technique used for scheduling instructions.

Processors that use *static scheduling* [16] check for data and structural hazards at decode time. The issue of the instruction is delayed in case there is a hazard, leading to an in-order issue and

execution of instructions. Static scheduling depends on software and compiler techniques to schedule instructions for minimizing such stalls. On the other hand, *dynamic scheduling* [16] decouples the decoding of an instruction from its issue and execution, with the execution unit assuming the responsibility for detecting structural and data hazards. The out-of-order issue and execution of instructions allows the processor to continue decoding and executing instructions even though some previous instructions may be delayed due to data dependences or busy functional units.

Most current commercial microprocessors are statically scheduled. Load and store instructions are often handled in a special manner, however. Stores are normally placed in a store buffer and the store buffer issues the store whenever the memory subsystem can accept another access. The processor can continue executing instructions even though there may be stores outstanding, thus overlapping the write latency with other computation and accesses. This has been shown to be an effective way to hide the latency of writes in multiprocessors [7]. Load instructions, on the other hand, are scheduled in one of two ways. The most common technique is to stall the processor for the return value. This removes the need for checking data dependence later. The second method simply issues the load, delaying the stall up to the next instruction that actually uses the return value. The former technique disallows any type of overlap with future instructions for the purposes of masking the latency of the read. Although the latter technique allows the processor to continue past the load, it is likely that the processor will stall within a few instructions due to data dependence, unless the compiler can successfully schedule instructions to avoid such a stall.

Dynamic scheduling provides the opportunity for the processor to continue past an outstanding load and the instructions that are dependent on the return value. The use of dynamic scheduling to hide memory latency dates back to the IBM Stretch [5] and IBM 360/91 [36]. In order to be effective in hiding latencies in the range of tens to hundreds of cycles, dynamic scheduling needs to be complemented with *register renaming* [19], *dynamic branch prediction* [23], and *speculative execution*. Register renaming alleviates write-after-read (WAR) and write-after-write (WAW) dependences that in turn may delay the execution of future instructions. Dynamic branch prediction increases the lookahead capability of the processor while speculative execution allows the processor to execute instructions past unresolved branches. Several of these techniques have been proposed for use in dynamically scheduled superscalar designs for multiple issue of instructions (e.g. [18, 27]). However, in this paper, we are mainly concerned with the use of these techniques for hiding memory latency. The degree to which these techniques succeed in hiding the latency of reads is determined by the size of the lookahead buffer from which the processor chooses instructions to execute, the predictability of branches, and finally the data dependence characteristics of the program, which determines the degree of independence among instructions. Our experimental results will shed more light on the effect of each of these factors on performance.

Overlapping memory accesses to hide latency also imposes several requirements on the memory system. In general, these requirements involve providing high bandwidth and avoiding serialization points. Let us first consider the memory subsystem within the processor environment. To effectively hide the latency of writes, the store buffer needs to allow reads to bypass pending writes. In addition, it is important for the caches to be lockup-free [21] in order to allow the servicing of reads while previous writes are being serviced. Overlap of multiple reads clearly extends the requirement on the cache to allow simultaneous service of reads also. Regarding the interconnection network and memory modules, it is important to allow accesses that are issued in an overlapped manner to be serviced concurrently. This requires sufficient bandwidth and the need to avoid serialization points within the interconnection network and at the memory modules.

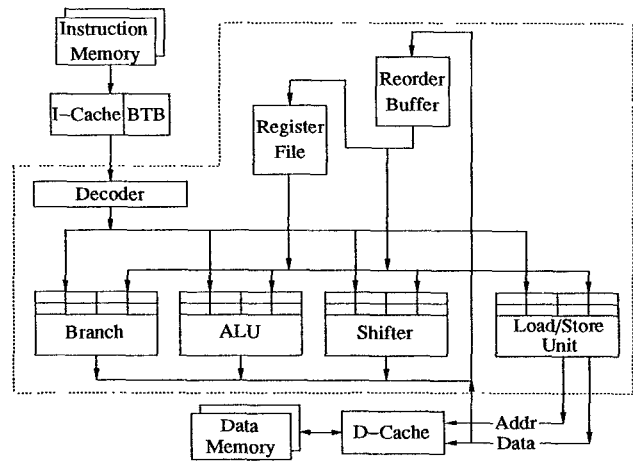


Figure 2: Overall structure of Johnson's dynamically scheduled processor.

# 3 Simulation Methodology

This section presents the different components used in our simulation environment. The first subsection discusses architectural details of the processor. The second subsection explains the coupling of the multiprocessor simulation with the processor simulator. Finally, the last subsection presents the benchmark applications used in our study.

## 3.1 Simulated Processor Architecture

The base processor used in our study is derived from an architecture proposed by Johnson [18]. Only a brief description is provided here; the interested reader is referred to [18] for more detail.

Figure 2 shows the overall structure of the processor. The processor consists of a number of functional units with associated *reservation stations* [36]. Although not shown in the figure, there are another four floating point units for performing floating point add, multiply, divide, and conversion.[1] The reservation stations are instruction buffers that decouple instruction decoding from instruction execution and allow for dynamic scheduling of instructions. Decoded instructions and operands are placed into the appropriate reservation station by the decoder. The reservation station can issue an instruction as soon as the instruction has its data dependences satisfied and the functional unit is free.

The reorder buffer [34] shown in Figure 2 is responsible for several functions in this architecture. The first function is to eliminate storage conflicts through register renaming [19]. Each instruction that is decoded is dynamically allocated a location in the reorder buffer and a tag is associated with its result register. The tag is updated to the actual result value once the instruction completes. Correct execution is achieved by providing the value or tag in the reorder buffer (instead of the value in the register file) to later instructions that attempt to read this register. Unresolved operand tags in the reservation stations are updated when the corresponding instruction completes.

[1] While Johnson assumes the floating point units reside on a coprocessor, we assume they are on the same chip as the integer units.

The second function for the reorder buffer is to provide the rollback mechanism necessary for supporting speculative execution past unresolved branches. The architecture uses a branch target buffer (BTB) [23] to provide dynamic branch prediction. The rollback mechanism for mispredicted branches works as follows. Decoded instructions enter the reorder buffer in program order. When an instruction at the head of the buffer completes, the entry belonging to it is deallocated and the result value is written to the register file. Since instructions enter the reorder buffer in program order and are retired in the same order, updates to the register file take place in program order. Therefore, instructions that depend on an unresolved branch are not committed to the register file until the branch completes. Similarly, memory stores are also held back (the reorder buffer controls the retiring of stores from the store buffer).[2] If a branch is mispredicted, all instructions past the branch are invalidated from the reorder buffer, the reservation stations, and the appropriate buffers, and decoding and execution is started from the correct branch target. The mechanism provided for handling mispredicted branches is also used to provide precise interrupts. This allows the processor to restart quickly without the need to save and restore complex state information.

For our study, we have assumed a single cycle latency for all functional units except the load/store unit. The latency for loads and stores is assumed to be one cycle in case of a cache hit. The scheduling of instructions within each functional unit is allowed to be out-of-order. This pertains to the load and store unit also as long as the memory consistency constraints allow it. Furthermore, for consistency models that allow it, load operations can bypass the store buffer and dependence checking is done on the store buffer to assure a correct return value for the load.

Regarding the caches, we simulate a lockup-free data cache [21] that allows for multiple outstanding requests. We ran our simulations assuming a single cache port, thus limiting the number of loads and stores executed to at most one per cycle. The branch target buffer size was set to 2048 entries with a 4-way set-associative organization. For simplicity, all instructions are assumed to hit in the instruction cache.

The next section describes how our processor simulator is coupled to the multiprocessor simulation.

## 3.2 Multiprocessor Simulation

Our multiprocessor simulation is done using Tango Lite, the lightweight thread version of the Tango simulation environment [11]. The simulation assumes a simple multiprocessor architecture. The architecture consists of 16 processors, each with a 64K data cache that is kept coherent using an invalidation-based scheme. The caches are direct mapped, write-back caches with a line size of 16 bytes. The simulated processors are simple in-order issue processors with blocking reads. Writes are placed in a write buffer. The consistency model assumed for the multiprocessor simulation is release consistency, thus the latency arising from writes is effectively hidden. The latency of memory is assumed to be 1 cycle for cache hits and a fixed number of cycles for cache misses. In our experiments, we use a 50 cycle penalty for cache misses (results for a 100 cycle penalty are provided in [9]). Queuing and contention effects in the interconnection network are not modeled.

The above simulation generates a dynamic instruction trace for each of the simulated processes. The generated trace is augmented with other dynamic information including the effective address for load and store instructions and the effective latency for each memory and synchronization operation.

[2] A store at the head of the reorder buffer is retired as soon as its address translation completes and the consistency constraints allow its issue. The store buffer is free to issue the store after this point.

To simulate the effects of dynamic scheduling, we choose the dynamic instruction trace for one of the processes from the multiprocessor simulation and feed it through our processor simulator (described in the previous section). Since the pertinent information about the multiprocessor cache simulation and the synchronization behavior is already contained in the trace, the processor simulator simply uses this information to determine the latency of memory accesses and synchronization.

The use of trace-driven simulation in our study may introduce some inaccuracies in our results [20]. This is because the trace is generated assuming a different processor architecture than the processor we eventually simulate. Thus, although the exact global interleaving of shared accesses may be different given the two processor architectures, we use the same ordering for both. The extent to which this affects results depends both on the application behavior and on the characteristics being studied. For our study, the program characteristics that determine how well dynamically scheduled processors hide memory latency are the cache miss behavior, the data dependence characteristics between memory accesses and computation, and the predictability of branches. We do not expect these characteristics to vary greatly due to slightly different global interleaving of the accesses.

One characteristic that can vary greatly due to different interleavings is the synchronization latency and load balance in an execution. Thus, synchronization times need to be treated cautiously in trace-driven simulations. In this study, we mainly focus on the fraction of read latency that can be hidden and the overhead of synchronization does not fundamentally affect the ability of dynamic scheduling techniques in hiding this latency. Synchronization times are only reported to provide an estimate for the fraction of execution time that corresponds to memory latency. Overall, we believe that our results are only minimally affected by the use of traces in our simulations.

## 3.3 Benchmark Applications

The applications chosen for this study are representative of algorithms used in an engineering and scientific computing environment. The five applications that we study are MP3D, LU, PTHOR, LOCUS, and OCEAN. The first four applications are written in C. OCEAN is written in Fortran. The applications use synchronization primitives provided by the Argonne National Laboratory macro package [4]. The first three applications have already been used in a previous study on consistency models [7]. Except for LU, the other four applications are part of the SPLASH suite from Stanford [32].

MP3D [26] is a 3-dimensional particle simulator used to study the pressure and temperature profiles created by an object flying at high speed through the upper atmosphere. The overall computation of MP3D consists of evaluating the positions and velocities of molecules over a sequence of time steps. During each time step, the molecules are picked up one at a time and moved according to their velocity vectors. Collisions of molecules among themselves and with the object and the boundaries are all modeled. The simulator is well suited to parallelization because each molecule can be treated independently at each time step. The main synchronization consists of barriers between each time step. For our experiments we ran MP3D with 10,000 particles in a 64x8x8 space array, and simulated 5 time steps.

LU performs LU-decomposition for dense matrices. The primary data structure in LU is the matrix being decomposed. Columns are statically assigned to the processors in an interleaved fashion. Each processor waits for the current pivot column, and then uses that column to modify all the columns that it owns. The processor that produces the current pivot column releases any

processors waiting for that column. For our experiments we performed LU-decomposition on a 200x200 matrix.

PTHOR [35] is a parallel distributed-time logic simulator based on the Chandy-Misra simulation algorithm. The primary data structures associated with the simulator are the logic elements (e.g., AND-gates, flip-flops), the nets (wires linking the elements), and the task queues which contain activated elements. Each processor executes the following loop. It removes an activated element from one of its task queues and determines the changes on that element's outputs. It then schedules the newly activated elements onto the task queues. For our experiments we simulated five clock cycles of a small RISC processor consisting of the equivalent of 11,000 two-input gates.

LOCUS [30] is a commercial quality VLSI standard cell router. The program evaluates standard cell circuit placement with the objective of finding an efficient routing that minimizes area. The main data structure is a cost array that keeps track of the number of wires running through each routing cell of the circuit. The parallelism exists at many levels. The many wires in a circuit can be routed in parallel. The routing of each wire can also be done in parallel since it involves determining paths for all independent two-pin segments of the wire. Finally, the several routes for finding the best path for each two-pin segment can be evaluated in parallel. For our experiments we used a circuit with 1266 wires and a 481-by-18 cost array.

OCEAN [31] models the role of eddy and boundary currents in influencing large-scale ocean movements. The simulation is performed for many time-steps until the eddies and mean ocean flow attain a mutual balance. The work in each time-step involves solving a set of spatial partial differential equations. The principle data structure consists of a set of two-dimensional arrays holding discretized values of the various functions associated with the model's equations. The two-dimensional structures are statically allocated as 200-by-200 double precision floating point arrays, and there are approximately 25 such arrays in the program. For our experiments we simulated a 98-by-98 point square grid.

Table 1 provides some statistics on data references for the five applications. This data is from the Tango Lite multiprocessor simulation assuming memory latency of 50 cycles. The numbers are for a single processor in the 16 processor simulation. The column marked busy cycles in Table 1 specifies the number of useful cycles (or instructions) in the application for a single processor. We should point out that the 64K data caches (per processor) are large relative to the problem sizes we are simulating. Thus, the cache misses reported mainly reflect inherent communication misses. Table 2 provides similar information about the synchronization references (wait event and set event implement general event synchronization for producer/consumer interactions). Since branch behavior is an important factor in determining the performance of dynamically scheduled processors, we also provide some statistics on branches in Table 3.

# 4 Simulation Results

This section studies the effect of relaxing the consistency model in conjunction with dynamically scheduled processors. Section 4.1 presents simulation results comparing statically and dynamically scheduled processors given different consistency models. The above results assume single instruction issue processors. Section 4.2 briefly discusses the effect of higher latency and using multiple instruction issue (simulation results for Section 4.2 appear in the full version of the paper [9]).

Table 1: Statistics on data references. Numbers are for a single processor in the 16 processor simulation. Numbers in parentheses are rates given as references per thousand instructions.

| *Program* | *Busy Cycles* | *reads (×1,000)* | *writes (×1,000)* | *read misses (×1,000)* | *write misses (×1,000)* |
|---|---|---|---|---|---|
| MP3D | 550,000 | 127 (230) | 63 (114) | 13.4 (24.3) | 12.4 (22.5) |
| LU | 2,277,000 | 697 (306) | 345 (151) | 16.4 (7.2) | 5.5 (2.4) |
| PTHOR | 1,758,000 | 701 (399) | 147 (83) | 41.2 (23.5) | 15.2 (8.7) |
| LOCUS | 3,325,000 | 697 (210) | 180 (54) | 30.9 (9.3) | 18.4 (5.5) |
| OCEAN | 5,141,000 | 1,550 (302) | 585 (114) | 112 (21.7) | 202 (39.3) |

Table 2: Statistics on synchronization. Numbers are for a single processor in the 16 processor simulation. Numbers in parentheses are rates given as references per thousand instructions.

| *Program* | *locks* | *unlocks* | *wait event* | *set event* | *barriers* |
|---|---|---|---|---|---|
| MP3D | 40 (0.07) | 40 (0.07) | 0 | 0 | 30 (0.05) |
| LU | 0 | 0 | 199 (0.09) | 13 (0.00) | 2 (0.00) |
| PTHOR | 6,038 (3.43) | 6,039 (3.44) | 134 (0.08) | 0 | 249 (0.14) |
| LOCUS | 356 (0.11) | 356 (0.11) | 2 (0.00) | 0 | 1 (0.00) |
| OCEAN | 21 (0.00) | 21 (0.00) | 0 | 0 | 150 (0.03) |

## 4.1 Static versus Dynamic Scheduling

Figure 3 shows the simulation results comparing statically and dynamically scheduled processors with variation of the memory consistency model. The left-most column in each graph shows the breakdown of the execution time on an in-order execution processor (BASE) which completes each operation before initiating the next one (i.e., no overlap in execution of instructions and memory operations). The bottom section of this column represents busy time or useful cycles executed by the processor. The black section above it represents the time that the processor is stalled waiting for acquire synchronization (i.e., locks, wait events, barriers). The two sections on top of this show the contribution of read miss and write miss latencies to the execution time. Release operations are included in the total write miss time. The penalty for cache misses is assumed to be 50 cycles in these simulations.

The next three groups of columns in each graph correspond to varying the consistency model. The models evaluated are sequential consistency (SC), processor consistency (PC), and release consistency (RC). For each model, we evaluate three different processor architectures. The first is a simple statically scheduled processor with blocking reads (SSBR). The second is a statically scheduled processor with non-blocking reads (SS). This processor is allowed to execute past read misses and the stall is delayed up to the first use of the return value. We assume a 16 word deep write buffer for the above processors. The SS processor also has a 16 word deep read buffer. Finally, we evaluate the dynamically scheduled processor (DS) described in Section 3.1. To allow us to better isolate the effects on hiding memory latency, we have limited the decode and issue rate for the DS processor to a maxi-

Table 3: Statistics on branch behavior.

| Program | Percentage of Instructions | Avg. Distance bet. Branches (in instructions) | Percentage Correctly Predicted | Avg. Distance bet. Mispredictions (in instructions) |
|---|---|---|---|---|
| MP3D | 6.1% | 16.4 | 90.8% | 176.9 |
| LU | 8.0% | 12.5 | 98.0% | 618.1 |
| PTHOR | 15.3% | 6.5 | 81.2% | 34.7 |
| LOCUS | 15.6% | 6.4 | 92.1% | 81.6 |
| OCEAN | 6.0% | 16.6 | 97.9% | 778.9 |

mum of 1 instruction per cycle. The size of the reorder buffer (or lookahead window) used is denoted at the bottom of each column. This size corresponds to the maximum number of instructions that can reside in the window at any given time. The window size is varied from 16 to 256 for the RC model. For SC and PC, we only show the results for the most aggressive buffer size since the gains from dynamic scheduling are relatively small for these models. All simulations assume an aggressive memory system with lockup-free caches, write buffers that allow reads to bypass writes, and the ability to issue one access per cycle from each node. Of course, the extent to which these features are exploited depends on the consistency model and the processor architecture.

Some general observations that can be made from the simulation results are: (i) SC does not allow the read and write latency to be hidden regardless of the processor architecture; (ii) PC is in general successful in hiding the latency of writes with statically scheduled processors and does not gain much from the use of dynamic scheduling; (iii) RC fully hides the latency of writes under static scheduling and can substantially reduce the read latency given dynamically scheduled processors with large window sizes. Section 4.1.1 briefly discusses the results for static scheduling. Sections 4.1.2 and 4.1.3 analyze the results for dynamic scheduling in greater detail.

#### 4.1.1 Static Scheduling Results

In this section, we briefly discuss the results for the two statically scheduled processors (SSBR and SS). We first concentrate on the results for the SSBR processor. The effect of consistency models on statically scheduled processors with blocking reads (SSBR) has been previously studied [7], and our results simply reconfirm the findings in the previous study. Thus, we simply state some of the interesting results from our simulation and refer the interested reader to the previous study for a more in-depth analysis. As can be seen in Figure 3, the SC model is not successful in hiding the latency of either reads or writes. The PC model, on the other hand, is in general successful in hiding the latency of writes by allowing reads to bypass previous writes. This latency is fully hidden for three of the applications (MP3D, LU, and PTHOR). The small amount of write latency in LOCUS is mainly due to the bursty nature of writes in the application which results in the write buffer filling up and stalling the processor once in a while (4% of the cycles). OCEAN is different since PC is only successful in hiding part of the write latency and a substantial amount remains. The reason arises from the fact that the number of write misses is substantially larger than the number of read misses in this application. Normally, the fact that the processor stalls on read misses gives the write buffer sufficient time to retire previous writes without filling up. However, this condition does not hold if the total latency due to write misses exceeds the latency from read misses [7]. In OCEAN, the write buffer stalls the processor due to being full for about 29% of the cycles under PC. The results for RC are as expected. RC successfully hides the latency of writes in all the applications and the problems that arise with PC are not present since RC allows writes to be overlapped, thus reducing the likelihood of the write buffer filling up.

We now consider the results for the SS processor. The difference between SS and SSBR is that SS does not block on read accesses and delays this stall until the first use of the return value. Therefore, there is the potential for hiding the latency of reads in the region between the read access and its first use. However, as the results indicate, the improvement over SSBR is minimal. This is mainly because the object code has not been rescheduled to exploit non-blocking reads (by moving the load access further away from the use of the return value). Thus, the processor is effectively stalled a short distance away from a read miss since the use of the return value is normally within a few instructions away. In the future, we plan to study the effect of compiler rescheduling to exploit non-blocking reads as an alternative to using dynamic scheduling techniques in hardware.

#### 4.1.2 Dynamic Scheduling Results

We begin by building some intuition for the different factors that affect the performance of dynamically scheduled processors, specifically in their ability to hide read latencies. We then analyze the results based on this intuition. Section 4.1.3 presents more simulation results to further isolate the factors that influence performance.

Three factors determine the effectiveness of dynamic scheduling in hiding memory latency. Two of these factors relate to the characteristics of the application: (i) data dependence behavior and (ii) control (or branch) behavior. The third factor is the size of the lookahead window (or reorder buffer) provided by the architecture.

The data dependence characteristics of an application is the most fundamental factor influencing performance. Dynamic scheduling depends on the presence of independent accesses and computation for hiding memory latency. For the applications we are considering, independent operations are present at different granularities. Typically, the applications exploit parallelism at the task or loop iteration level and each processor is assigned a set of tasks or loop iterations. One source of independent operations is within the task or loop iteration boundary. Another source of independence is operations from two different tasks or loop iterations that have been assigned to the same processor. The availability of such independent operations and the distance between them determines the feasibility of a hardware technique in finding and overlapping them.

The behavior of branches in an application is also a key factor that affects performance. The frequency of branches and their predictability determines whether it is feasible for hardware to find distant independent operations.

Finally, the size of the lookahead window determines the maximum distance between independent operations that can be overlapped by the hardware. Therefore, to overlap two independent accesses, the size of the lookahead window needs to be at least as large as the distance between the two accesses. Furthermore, to fully overlap the latency of memory accesses with computation requires enough independent instructions to keep the processor busy for the duration of the latency. This requires the window size to be at least as large as the latency of memory. For example, with the latency of 50 cycles, we require a window size of at least 50 instructions. Smaller window sizes overlap only a fraction of this latency (proportional to the size).

We now consider the results with dynamically scheduled processors for the SC and PC models (Figure 3). Under SC, the implementation delays each access until the previous access is completed. Therefore, it is likely that the processor will stall

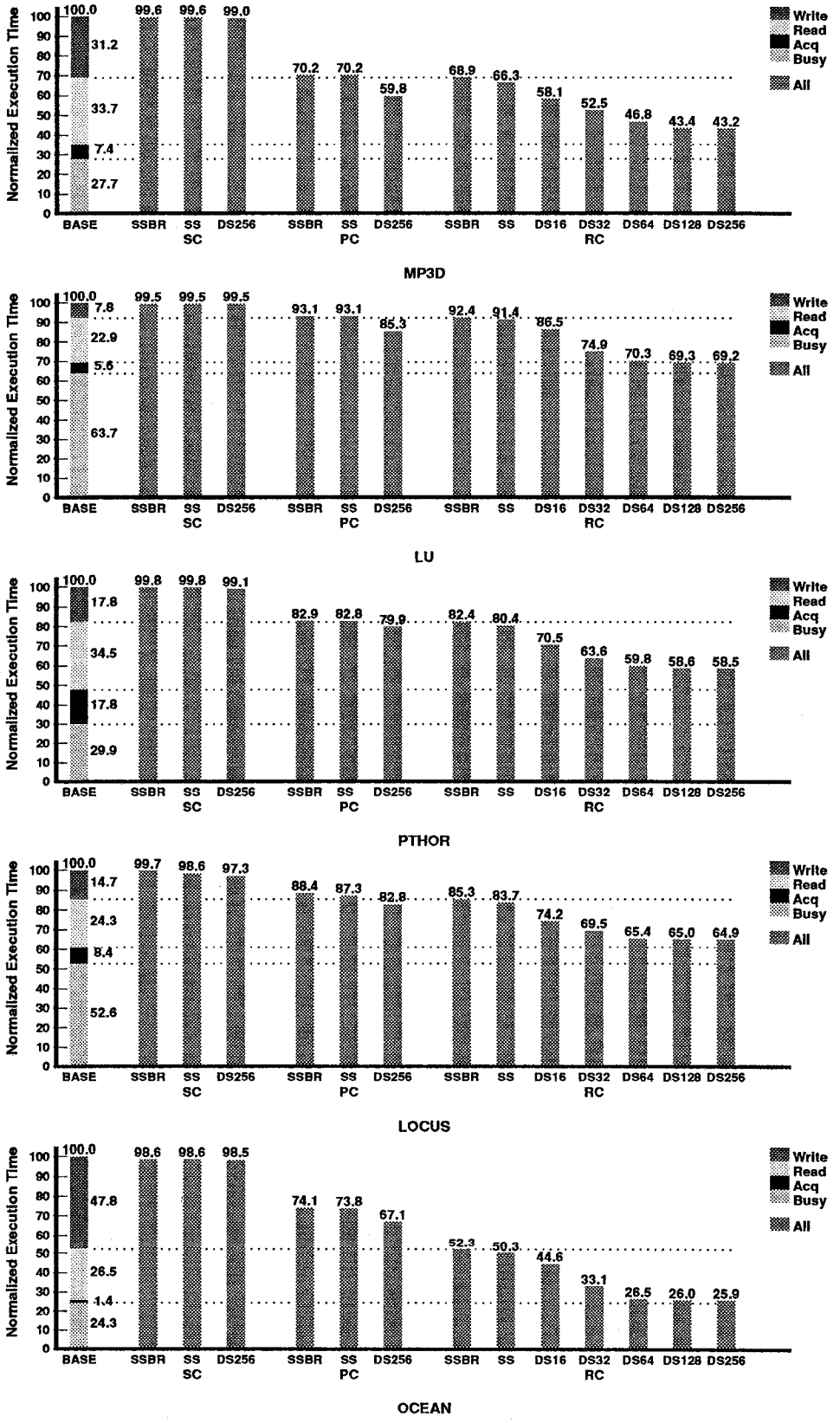


Figure 3: Simulation results for memory latency of 50 cycles.

shortly after a read or write miss due to its inability to complete future reads. As expected, the results for SC with the aggressive window size of 256 show virtually no improvement over static scheduling.

The PC model is slightly more relaxed than SC in that reads can bypass or overlap previous writes. This allows write latencies to be hidden. However, read accesses are still serialized. Due to this serialization, PC is in general not successful in hiding much of the read latency either, even with the aggressive window size of 256. The more noticeable gains in MP3D and LU most likely arise from the presence of independent computation on data that already resides in registers, which in turn allows the processor to continue past read misses for a short time without requiring another read value. As for OCEAN, the increase in performance compared to static scheduling mainly arises from the larger write buffer used for the DS processor which reduces the frequency of stalls due to a full write buffer.

Before analyzing the results for RC, let us first consider the bound on the gain achievable by dynamic scheduling given that read and write accesses can be overlapped between synchronization points. Since we limit issue to a maximum of one instruction per cycle, the time due to the computation (busy/useful time) cannot be diminished. In addition, acquire synchronization overhead arising from load imbalance or contention for synchronization variables is also impossible to hide with the techniques we are considering. On the other hand, the fraction of the synchronization overhead arising from the memory latency to access synchronization variables can be hidden in the same way that normal memory latency is hidden. For example, the latency to access a free lock can be hidden by overlapping this time with the computation prior to it. Relating this to the applications, our detailed simulation results show that virtually all of the acquire overhead in MP3D, LU, LOCUS, and OCEAN arises from load imbalance and contention. Therefore, the best dynamic scheduling can do for these applications is to fully hide the read and write (including release) latency. In PTHOR, however, approximately 30% of the acquire overhead is due to latency for accessing free locks. Therefore, in PTHOR, it is theoretically possible to hide a fraction of the acquire overhead as well.

Referring to the results for RC in Figure 3, we notice that there is a gradual increase in performance as we move from window size of 16 to window size of 256, with the performance increasing more rapidly for the small window sizes and leveling off at the larger windows. For LU and OCEAN, dynamic scheduling hides virtually all memory latency at window size of 64. For LOCUS, almost all latency, except for 16% of the read latency, is hidden at the larger window sizes. For MP3D, 24% of the read latency remains even at the larger window sizes. Similarly, PTHOR has 31% of its read latency remaining.

Data dependence and small window size effects are most likely the two factors that influence performance at the small window sizes (16 and 32). Based on the average distance between mispredicted branches shown in Table 3, branch behavior is not expected to be a determining factor for the small windows, except possibly in PTHOR. The factors shift at the larger window sizes. In LU and OCEAN, performance levels off at larger windows simply because the memory latency is virtually all hidden at the smaller window sizes. On the other hand, the leveling off of performance in MP3D, PTHOR, and LOCUS is most likely due to the fact that the branch prediction falls short of effectively using such large window sizes.

#### 4.1.3 Detailed Analysis of Dynamic Scheduling

This section provides simulation results to further isolate the effects of different factors influencing the performance of dynamically scheduled processors. Figure 4 presents these results for all five applications. The left-most column in each graph repeats the data presented in Figure 3 for the BASE processor. The next five columns in each graph provide performance results for the different window sizes assuming perfect branch prediction. The last five columns in each graph present the results given for the case where branch prediction is perfect and data dependences are being ignored.[3] Since the gains from dynamic scheduling were substantial only under the RC model, the results in this section all assume this model.

To isolate the effect of branches on performance, we can compare the performance for each window size with and without perfect branch prediction (left side of Figure 4 and right side of Figure 3). For LU and OCEAN, the branch prediction is already so good that we see virtually no gain from perfect branch prediction even at the largest window size of 256. For LOCUS, perfect branch prediction performs slightly better at the large window sizes of 128 and 256. For MP3D, noticeable gain from perfect branch prediction starts at window size of 64. PTHOR, which has the worst branch behavior of all the applications, seems to noticeably gain from perfect branch prediction even at the small window sizes of 16 and 32. Thus, except for PTHOR, the branch behavior of the applications does not seem to affect the performance until we reach the 128 and 256 window sizes.

To isolate the effect of data dependences, we can look at the improvement in performance at each window size when we move from perfect branch prediction to additionally ignoring data dependences (left and right side in Figure 4). For LU and OCEAN, there is little or no gain from ignoring data dependences, pointing to the fact that data dependences do not hinder performance in these two applications (this is as expected since it was possible to hide all memory latency for these two applications). For MP3D, PTHOR, and LOCUS, we observe that ignoring data dependences increases the performance more at the small window sizes. In fact, at window size of 256, the performance with and without ignoring data dependences is virtually the same in these three applications. This points to the fact that there are some data dependences at short distances (within a task or loop iteration), however, by looking ahead at substantially larger distances, it is possible to find more independent operations (across tasks or loop iterations). Although this is an interesting observation, from a practical point of view, it is not possible to exploit such independence at large distances with hardware techniques simply because the branch prediction falls short of providing such lookahead distances.

Our detailed simulation results also confirm the presence of data dependences at short distances. One such result measures the delay of each read miss from the time the instruction is decoded and placed in the reorder buffer to the time the read is issued to memory. Below, we consider this measure for the window size of 64 given perfect branch prediction. In LU and OCEAN, we find that read misses are rarely delayed more than 10 cycles. This clearly points to the fact that read misses are independent from one another in these two applications. Results for MP3D show that about 15% of the read misses are delayed over 40 cycles. Similarly, in LOCUS, more than 20% of read misses are delayed over 40 cycles. This indicates the presence of read misses that are data dependent on one another, with one read miss affecting the address of the next read miss. Data dependences have the most serious effect in PTHOR. For PTHOR, around 50% of the read misses are delayed over 50 cycles. This is indicative of dependence chains formed by multiple misses. One effect of such dependence chains is that there are fewer independent accesses to overlap. Furthermore, it is more difficult to fully hide the latency of read misses in such a chain by overlap with independent

[3] Dependences arising from consistency constraints, for example from an acquire to the following access, are still respected.

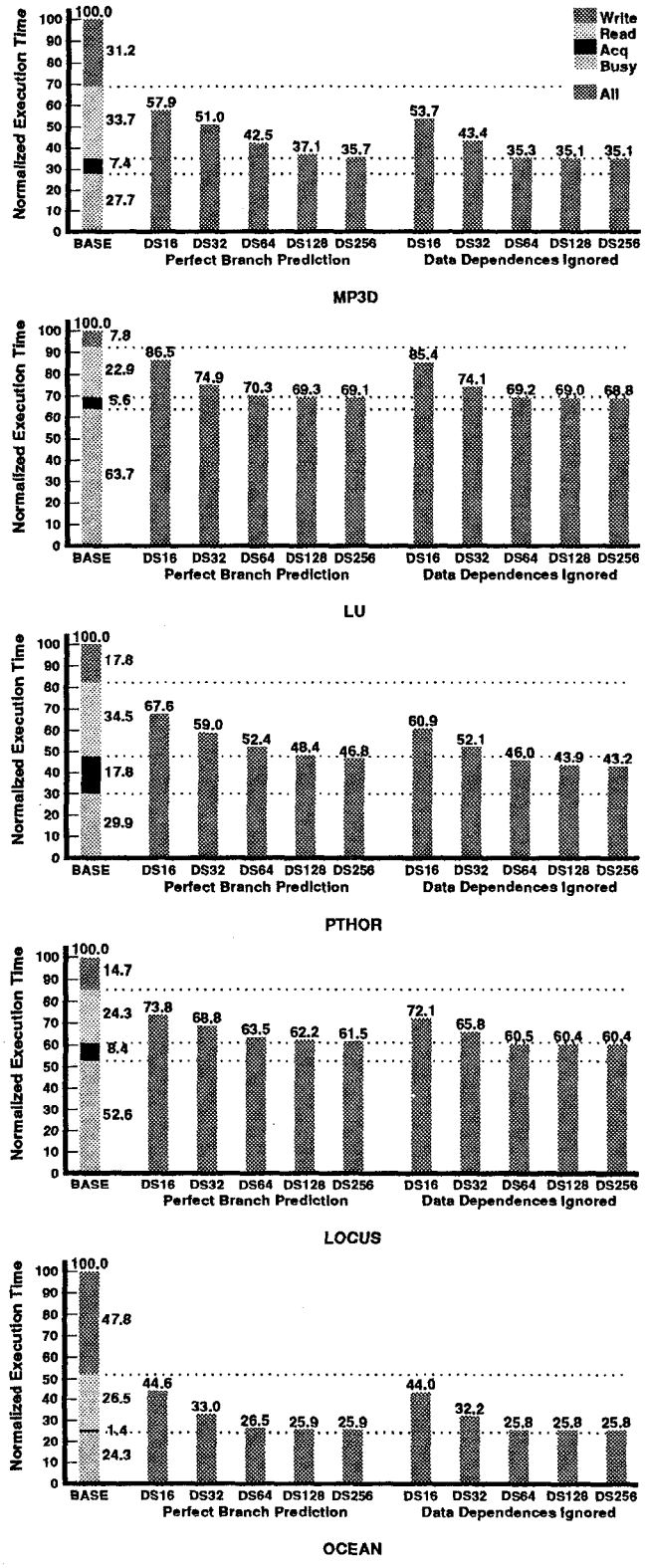


Figure 4: Effect of perfect branch prediction and ignoring data dependences for dynamic scheduling under RC.

computation, simply because the misses in the chain behave like a single read miss with double or triple the effective memory latency. This is the main reason why it takes window sizes of up to 256 to fully hide the latency of read misses in MP3D and PTHOR even with perfect branch prediction.

Finally, we can isolate the effect of window size by further analyzing the results for when both branch prediction is perfect and data dependences are ignored (right hand side in Figure 4). In this case, the only remaining factor that can affect performance is the window size. Looking at the results, we notice that window sizes of 16 and 32 do not fully hide the latency of reads. As explained in the previous section, the reasons for this are (i) small window sizes do not find independent operations that are farther apart than the window size, and (ii) to fully overlap latency with computation, the window size (in instructions) needs to be at least as large as the latency of access (in cycles). Relating to (i), our detailed simulation data for LU show that 90% of the read misses are a distance of 20-30 instructions apart. Thus, the window size of 16 performs relatively poorly compared to other applications. Similarly, in OCEAN, about 55% of the read misses are 16 to 20 instructions apart. Again, the reason for the poor performance at window size of 16 is that the distance between independent accesses exceeds the window size. Regarding reason (ii), we observe from the results that once we reach window size of 64 (which exceeds the memory latency), the latency can be fully overlapped with computation given the assumed lack of data dependences.[4]

In summary, data dependence and branch predictability do not hinder performance in LU and OCEAN. The performance of LOCUS and MP3D are primarily affected by data dependence at small window sizes and by branch predictability at large window sizes. Finally, PTHOR is affected by both data dependence and branch behavior at all window sizes.

## 4.2 Effect of Higher Latency and Multiple Instruction Issue

We briefly discuss the effect of higher memory latency and multiple instruction issue in this section. The simulation results for these experiments are presented in the full version of the paper [9].

We repeated the simulations assuming a higher memory latency of 100 cycles. The trends were similar to what we observed for the 50 cycle memory latency. The most obvious difference was that the performance leveled off at the window size of 128 instead of 64. This arises from the fact that the window size needs to exceed the latency in order to fully overlap memory latency with computation. Another observation was that the relative gain in performance from hiding memory latency was consistently larger for the higher latency of 100 cycles.

We also did a preliminary study of the effect of multiple instruction issue. In [9], we present some simulation results for issuing a maximum of four instructions per cycle. As with the higher latency results, the performance was still increasing when we moved from window size of 64 to 128 under RC, while without multiple issue, the performance virtually leveled off at window size of 64 (see Figure 3). This arises from the fact that multiple issue speeds up the computation, while memory latency remains at 50 cycles. Thus, a larger window size is needed to fully overlap the memory latency with computation time. Our preliminary results also indicated that the relative performance gains from multiple issue over

[4]Given perfect branch prediction and the assumed lack of dependence, the execution times for MP3D, LU, LOCUS, and OCEAN asymptotically approach the time for the computation plus the acquire overhead. For PTHOR, however, some of the acquire overhead is also being hidden. As explained in the previous section, the reason for this is that about 30% of the acquire overhead in PTHOR arises from latency for accessing free locks.

single issue were larger with RC than with SC. We are planning to do further simulations to better understand the interaction between multiple issue and hiding large memory latencies.

# 5 Discussion

Previous work has shown that relaxed models are effective in hiding latency arising from write accesses [7]. The goal of this study has been to evaluate relaxed models with respect to hiding read latency. We used a dynamically scheduled processor to exploit the overlap allowed by relaxed models for hiding this latency.

The choice of the application domain clearly influences our results. The scientific and engineering parallel programs used in this study typically exhibit sufficient instruction-level parallelism that can be exploited to hide memory latency. Furthermore, the branch predictability in these applications is usually better than for most uniprocessor programs.

Our architectural assumptions also influence the results. Our results are somewhat optimistic since we assume a high bandwidth memory system. In addition, we do not model the effect of contention in the network or at memory modules. However, the results are pessimistic in that we did not consider the effect of compiler help to schedule code to allow the processor to use the lookahead window more effectively. In addition, the FIFO retirement of instructions from the lookahead window (to provide precise interrupts) is a conservative way of using the window since instructions that have already been executed may remain in the window, disallowing new instruction from being considered for overlapped execution. Other techniques for providing precise interrupts may allow for better use of the window space. More aggressive branch prediction strategies may also allow higher performance for the applications with poor branch prediction. Finally, our simulation results are based on relatively large memory latencies. Smaller memory latencies will require proportionally smaller window sizes to achieve good performance.

The factors that determine the viability of dynamically scheduled processors are numerous. The most important concern about such processors is the extra hardware complexity and its effect on cycle time. These factors have to be considered in determining the actual gain from dynamic scheduling. In addition, other techniques for hiding latency need to be considered. For example, the overlap of memory accesses allowed by relaxed models can also be exploited by the compiler for scheduling read misses to mask their latency on a statically scheduled processor with non-blocking reads. Other competitive techniques for hiding latency include the use of multiple contexts [2, 15, 17, 33, 37] and prefetching [13, 24, 28, 29]. Choosing among these different techniques requires a careful study of the tradeoffs of each technique [14]. In fact, some combination of these techniques may be the most viable choice.

# 6 Related Work

This section discusses the related work on hiding memory latency. We begin by describing related work in the context of uniprocessors and then discuss the work directly related to shared-memory multiprocessors.

Baer and Chen [3] have suggested a dynamic hardware technique for uniprocessors that prefetches accesses into the cache before they are used. The technique depends on dynamically detecting regular access patterns and issuing prefetches before the processor issues the access. Although this scheme may achieve reasonable gains for applications with regular access behavior (e.g., LU and OCEAN), the technique would probably fail to hide latency for applications that do not have such regular characteristics (e.g., MP3D, PTHOR, LOCUS).

Melvin and Patt [27] have observed that dynamically scheduled processors are less sensitive to higher memory latency than statically scheduled processors. Their study was done in the context of multiple issue of instructions in uniprocessors and the largest cache miss penalty used was 10 cycles. In addition, the behavior (e.g., cache miss, data dependence, and branch behavior) of the uniprocessor applications they studied is substantially different from the parallel applications used in our study.

Lee et al. [25] have studied the effect of a hardware-controlled prefetching scheme in a multiprocessor architecture with software-based cache coherence. They use sixteen subroutines to evaluate the gains from prefetching assuming a memory latency of 20 cycles and instruction lookahead buffers of up to 16 entries. Since caches are kept coherent by software, any shared access that cannot be analyzed by the compiler is marked as non-cacheable. Lee et al. conservatively treat these non-cacheable accesses as synchronization, effectively imposing consistency constraints at these points. Thus, the performance of their scheme depends heavily on whether the compiler can identify enough shared accesses as cacheable in order to avoid the serialization that occurs at non-cacheable accesses. Since the compiler is usually conservative, we expect the serialization due to non-cacheable accesses to have hindered the ability to achieve full overlap among accesses (as is allowed in a relaxed model such as RC). Their results show that the performance gain is indeed mainly limited due to the presence of such serialization points and in some cases due to poor branch prediction.

Regarding the role of memory consistency models in hiding memory latency in multiprocessors, two new techniques have been recently proposed for boosting the performance of sequential consistency relative to other relaxed models [8]. These techniques aggressively overlap memory accesses without violating SC. The first technique involves a non-binding prefetch of values for the accesses that are delayed due to consistency constraints. The second technique employs speculative execution on read values to allow the processor to proceed with reads even though the consistency constraints disallow this in general (there is rollback ability if a violation of SC is detected). The degree to which these techniques boost the performance of strict consistency models remains to be fully studied. While the success of these techniques may deemphasize the correctness aspect of overlapping memory accesses in multiprocessors at the hardware level, the underlying mechanisms that enable and exploit such overlap still remain important.

# 7 Concluding Remarks

Relaxing the consistency model has been proposed as an attractive way to hide the large memory latency in shared-memory multiprocessors. A previous study evaluated the performance of relaxed models in the context of invalidation-based cache coherent multiprocessors [7]. The results of the study showed that relaxed models such as processor consistency and release consistency are effective in fully hiding the latency of writes. However, the opportunity to overlap read accesses, as allowed by models such as release consistency, was not exploited due to the use of statically scheduled processors with blocking reads. Therefore, the effect of relaxed models on hiding the latency of reads remained unexplored.

In this paper, we have explored the performance of relaxed models for hiding read latency using dynamically scheduled processors. The study was based on simulation results using five scientific and engineering parallel applications. Assuming a mem-

ory latency of 50 cycles, the average percentage of read latency that was hidden across the five applications was 33% for window size of 16, 63% for window size of 32, and 81% for window size of 64. For two of the applications, LU and OCEAN, read latency was fully hidden at the 64 window size. In general, larger window sizes did not increase the performance substantially. The trends for a memory latency of 100 cycles were similar, except that larger windows were needed to fully hide the read latency. In isolating the different factors that affect performance, we identified data dependences at short distances and small lookahead as the limiting factors for small windows and branch prediction as the limiting factor for large windows.

These results show that a substantial fraction of the memory latency can be hidden by overlapping accesses as allowed by relaxed models. However, whether the use of dynamically scheduled processors is viable for exploiting this overlap in hardware depends more on the feasibility of building such processors, especially with large window sizes. As future research, it would be interesting to evaluate compiler techniques that exploit relaxed models to schedule reads early. Such compiler rescheduling may allow dynamic processors with small windows or statically scheduled processors with non-blocking reads to effectively hide read latency with simpler hardware.

# 8 Acknowledgements

We thank the anonymous reviewers for their comments. We also thank the following people for providing useful comments on an earlier version of the paper: Sarita Adve, Phillip Gibbons, Michael Smith, Per Stenstrom, and Rick Zucker. We greatly appreciate the help of several people who made it possible to do the simulations. Stephen Goldschmidt provided us with the Tango Lite environment. The dynamically scheduled processor simulator was based on a simulator previously developed at Stanford by Mike Johnson. We thank him and Michael Smith for helping us extend the simulator for our experiments. We thank the application writers for their applications: Jeff McDonald for MP3D, Ed Rothberg for LU, Larry Soule for PTHOR, Jonathan Rose for LOCUS, and Jaswinder Pal Singh for OCEAN. Charles Orgish was instrumental in providing us with enough workstations to complete the simulations in a timely fashion. Finally, we thank Rohit Chandra for the many useful discussions.

This research was supported by DARPA contract N00039-91-C-0138. Kourosh Gharachorloo is partly supported by Texas Instruments. Anoop Gupta is partly supported by a NSF Presidential Young Investigator Award.

# References

[1] Sarita Adve and Mark Hill. Weak ordering - A new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[2] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. April: A processor architecture for multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, May 1990.

[3] Jean-Loup Baer and Tien-Fu Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of Supercomputing '91*, pages 176–186, November 1991.

[4] James Boyle et al. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc., 1987.

[5] W. Buchholz, editor. *Planning a Computer System: Project Stretch*. McGraw-Hill, 1962.

[6] Michel Dubois, Christoph Scheurich, and Fayé Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, June 1986.

[7] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245–257, April 1991.

[8] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Two techniques to enhance the performance of memory consistency models. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages I:355–364, August 1991.

[9] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Hiding memory latency using dynamic scheduling in shared-memory multiprocessors. Technical report, Stanford University, April 1992.

[10] Kourosh Gharachorloo, Dan Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[11] Stephen R. Goldschmidt and Helen Davis. Tango introduction and tutorial. Technical Report CSL-TR-90-410, Stanford University, 1990.

[12] James R. Goodman. Cache consistency and sequential consistency. Technical Report Computer Sciences #1006, University of Wisconsin, Madison, February 1991.

[13] E. Gornish, E. Granston, and A. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *International Conference on Supercomputing*, pages 354–368, 1990.

[14] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber. Comparative evaluation of latency reducing and tolerating techniques. In *Proceeding of the 18th Annual International Symposium on Computer Architecture*, pages 254–263, May 1991.

[15] Robert H. Halstead, Jr. and Tetsuya Fujita. MASA: A multithreaded processor architecture for parallel symbolic computing. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 443–451, June 1988.

[16] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

[17] R. A. Iannucci. Toward a dataflow/von Neumann hybrid architecture. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 131–140, June 1988.

[18] Mike Johnson. *Superscalar Microprocessor Design*. Prentice Hall, 1991.

[19] R. M. Keller. Look-ahead processors. *Computing Surveys*, 7(4):177–195, 1975.

[20] Eric J. Koldinger, Susan J. Eggers, and Henry M. Levy. On the validity of trace-driven simulation for multiprocessors. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 244–253, May 1991.

[21] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 81–85, 1981.

[22] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):241–248, September 1979.

[23] J. K. F. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, 17:6–22, 1984.

[24] Roland L. Lee. *The Effectiveness of Caches and Data Prefetch Buffers in Large-Scale Shared Memory Multiprocessors*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, May 1987.

[25] Roland L. Lee, Pen-Chung Yew, and Duncan H. Lawrie. Data prefetching in shared memory multiprocessors. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 28–31, August 1987.

[26] Jeffrey D. McDonald and Donald Baganoff. Vectorization of a particle simulation method for hypersonic rarified flow. In *AIAA Thermodynamics, Plasmadynamics and Lasers Conference*, June 1988.

[27] Stephen Melvin and Yale Patt. Exploiting fine-grained parallelism through a combination of hardware and software techniques. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 287–296, May 1991.

[28] Todd Mowry and Anoop Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[29] Allan K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Department of Computer Science, Rice University, May 1989.

[30] Jonathan Rose. Locusroute: A parallel global router for standard cells. In *Design Automation Conference*, pages 189–195, June 1988.

[31] Jaswinder Pal Singh and John L. Hennessy. Parallelizing the simulation of ocean eddy currents. Technical Report CSL-TR-89-388, Stanford University, August 1989.

[32] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. Technical Report CSL-TR-91-469, Stanford University, May 1991.

[33] Burton J. Smith. Architecture and applications of the HEP multiprocessor computer system. *SPIE*, 298:241–248, 1981.

[34] J. E. Smith and A. R. Pleszkun. Implementation of precise interrupts in pipelined processors. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 36–44, June 1985.

[35] Larry Soule and Anoop Gupta. Parallel distributed-time logic simulation. *IEEE Design and Test of Computers*, 6(6):32–48, December 1989.

[36] R. M. Tomasulo. An efficient hardware algorithm for exploiting multiple arithmetic units. *IBM Journal*, 11:25–33, 1967.

[37] Wolf-Dietrich Weber and Anoop Gupta. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 273–280, June 1989.