ABSTRACT

IBRAHIM, KHALED ZAKARYA. Slipstream Execution Mode for CMP-based Shared Memory Systems. (Under the direction of Dr. Gregory T. Byrd.)

Scalability of applications on distributed shared-memory (DSM) multiprocessors is limited by communication and synchronization overheads. At some point, using more processors to increase parallelism yields diminishing returns or even degrades performance. When increasing concurrency is futile, we propose an additional mode of execution, called *slipstream mode*, that instead enlists extra processors to assist parallel tasks by reducing perceived overheads.

We consider DSM multiprocessors built from dual-processor chip multiprocessor (CMP) nodes (e.g., IBM Power-4 CMP) with shared L2 cache. A parallel task is allocated on one processor of each CMP node. The other processor of each node executes a reduced version of the same task. The reduced version skips shared-memory stores and synchronization, allowing it to run ahead of the true task. Even with the skipped operations, the reduced task makes accurate forward progress and generates an accurate reference stream, because branches and addresses depend primarily on private data.

Slipstream execution mode yields multiple benefits. First, the reduced task prefetches data on behalf of the true task. Second, reduced tasks provide a detailed picture of future reference behavior, enabling a number of optimizations aimed at accelerating coherence events. We investigate a well-known optimization, self-invalidation. We also investigate providing confidence mechanism for speculation after barrier synchronization.

We investigate the implementation of an OpenMP compiler that supports slipstream execution mode. We discuss how each OpenMP construct can be implemented to take advantage of slipstream mode, and we present a minor extension that allows runtime or compile-time control of slipstream execution. We also investigate the interaction between slipstream mechanisms and OpenMP scheduling. Our implementation supports both static and dynamic scheduling in slipstream mode.

For multiprocessor systems with up to 16 CMP nodes, Slipstream mode is 12-

19% faster with prefetching only. With self-invalidation also enabled, performance is improved by as much as 29%.

We extended slipstream mode to provide a confidence mechanism for barrier speculation. This mechanism identifies dependencies and tries to avoid dependency violations that lead to misspeculations (and subsequently rollbacks). Rollbacks are reduced by up to 95% and the improvement in performance is up to 13%.

Slipstream execution mode enables a wide range of optimizations based on an accurate future image of the program behavior. It does not require custom auxiliary hardware tables used by history-based predictors.

Slipstream Execution Mode for CMP-based Shared Memory Systems

by

Khaled Z. Ibrahim

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial satisfaction of the
requirements for the Degree of
Doctor of Philosophy

Department of Electrical and Computer Engineering

Raleigh

2003

Approved By:

Dr. Eric Rotenberg	Dr. Frank Mueller
Dr. Gregory T. Byrd Chair of Advisory Committee	Dr. Thomas M. Conte

To my parents and my small family Dina, Sief, and Omar.

Biography

Khaled Z. Ibrahim was born in Port Said, Egypt. He received the B.Sc. and M.Sc. in computer engineering from Suez Canal University in 1993 and 1997, respectively. He worked as a system programmer for different software companies. He enrolled in the Ph.D. program in North Carolina State University in 1999. He was awarded the Ph.D. degree in 2003.

Acknowledgements

First, I would like to thank God Almighty for his help to complete this work.

I owe an immense debt to my advisor Dr. Gregory T. Byrd. Without his direction, I would not have got my thesis to reach this shape. Dr. Byrd provided me continuous encouragement and patient advice that empowered me to move ahead. He was deeply engaged with me from the formative stage, through the refining of this work, to the final writing. His deep understanding both technically and personally makes me feel always respectful and grateful to him.

I would like also to acknowledge the help I received from Dr. Eric Rotenberg who was deeply involved in this work. I always found his critics very enlightening. He helped in developing and refining many parts of this work.

For their positive critiques, I feel very thankful to my advisory committee, Dr. Thomas M. Conte and Dr. Frank Mueller.

I would like to the acknowledge the support this work received by the NSF Computer System Architecture program, contract CCR-0105628.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivations

Scalability for many parallel programs is limited by communication and synchronization overheads. A performance threshold is reached (for a fixed problem size), and applying more processors results in little or no speedup. The only means for moving beyond this threshold is to increase efficiency – to identify and remove bottlenecks and overheads and more effectively use the parallel computing resources.

In this work, we consider the use of a dual-processor chip multiprocessor (CMP) as the building block of a distributed shared memory multiprocessor. Examples of academic and commercial chip multiprocessors are the Stanford Hydra [38] and IBM Power-4 [20], respectively.

The philosophy behind putting multiple cores on one die is to extract more parallelism by executing more than one thread at once. Multiple threads are expected to provide more opportunities of parallelism that can keep more functional units busy, compared with having all the functional units in a unified processor. This design view continues the trend of concentrating more on compute-centric microarchitecture design. Communications is always being thought as a separate design issue. Although evolved in the era when computations were more expensive than communication. This view is still driving the microarchitecture industry today.

A conventional way to use a system composed of CMPs is to assign a parallel task

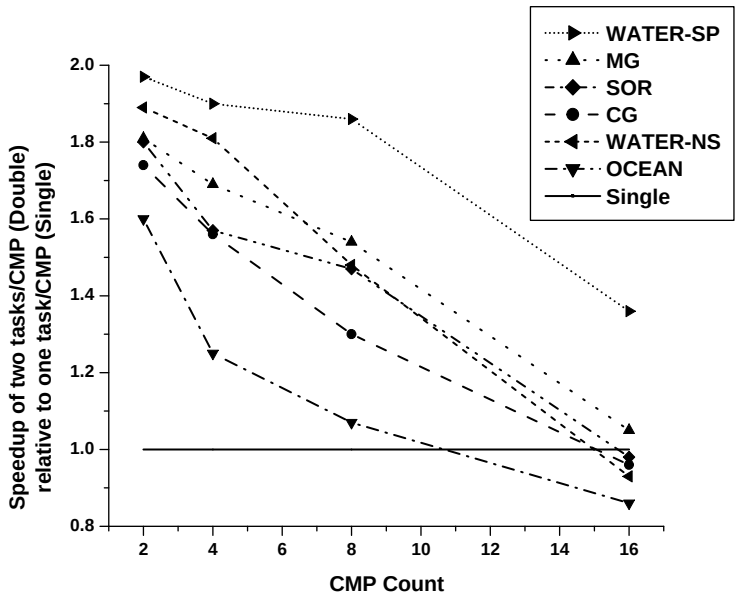


Figure 1.1: Relative speedup of two tasks per CMP (double mode) *vs.* one task per CMP (single mode).

to each processor. As we approach the performance threshold, however, increasing concurrency does not help. Figure 1.1 shows the relative performance of assigning two tasks per CMP (double mode), compared to assigning only a single task, leaving one processor idle (single mode). Applying the additional processing power in the "traditional" way – that is, by increasing the task-level parallelism – does not necessarily result in large performance gains, especially as the number of CMPs increases. In fact, for some applications, performance degrades when using both processors for parallel tasks. Figure 1.2 shows the execution time breakdown for Water-NS as an example for applications that suffer a performance degradation when task level parallelism is increased for 16 CMPs systems. We notice that the contribution of busy cycles to total execution time gets smaller when the degree of parallelism is increased. Overheads like synchronization and stall time also contribute more to the total execution time. This explains the need to devote more attention to these overheads for higher a degree of parallelism.

If increasing parallelism does not improve performance significantly, then we pro-

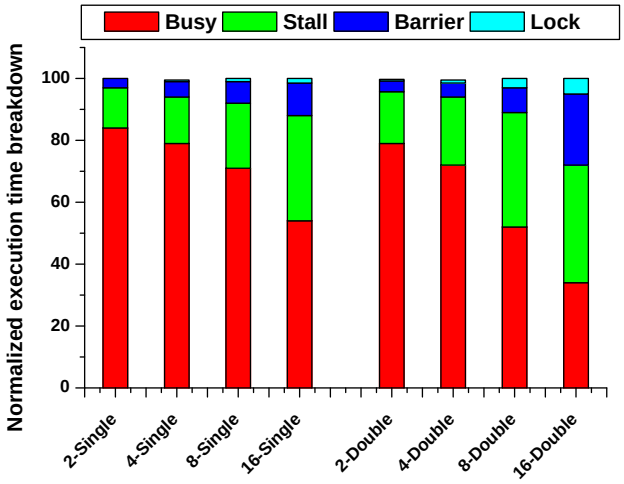


Figure 1.2: Execution time breakdown for Water-NS on 2, 4, 8, and 16 CMPs systems running in single and double modes.

pose using the second processor to reduce overhead and improve the efficiency of execution, rather than to increase concurrency. Instead of running a separate parallel task, the second processor runs a reduced version of the original task. The reduced task constructs an accurate view of future memory accesses, which is used to optimize memory requests and coherence actions for the original task. The resulting gains in efficiency can result in better performance than using the two processors for increased parallelism.

This approach is analogous to the uniprocessor slipstream paradigm [48], which uses redundant execution to speed up sequential programs. A slipstream processor runs two redundant copies of a program on a dual-processor CMP. A significant number of dynamic instructions are speculatively removed from one of the program copies, without sacrificing its ability to make correct forward progress. In this way, the speculative reduced program (called the *A-stream*, or advanced stream) runs ahead of the unreduced program (called the *R-stream*, or redundant stream). The R-stream exploits the A-stream to get an accurate picture of the future including both future values and branch outcomes.

In the multiprocessor setting, we do not need to remove a large number of dynamic

instructions. We find that simply removing certain long-latency events – synchronization events and stores to shared memory – shortens the A-stream version of a task enough to provide timely and accurate predictions of memory accesses for the original R-stream task. Since it runs ahead, the A-stream generates loads to shared data before they are referenced by the R-stream, prefetching shared data and reducing memory latencies for the R-stream. This simple use of redundant computation requires very little hardware support; it is an additional *mode of execution* for multiprocessor systems, rather than a new architecture.

With some additional support in the memory subsystem, A-stream accesses can also be used at the directory as hints of future sharing behavior. These hints can be used for coherence optimizations, such as *self-invalidation* (SI) [27]. To support SI, we introduce the notion of a *transparent load* to minimize negative effects of A-stream prefetches on coherence traffic. A transparent load, issued by the A-stream, does not cause the exclusive owner of a cache line to give up ownership prematurely. The load serves as an indication of future sharing behavior, and the memory controller uses this information to send invalidation hints to the exclusive owner of a cache line. The owner is advised to write back or invalidate its copy of the cache line when its last store is complete, so that future reads from other processors will find the most recent data in memory.

Using these two optimizations (prefetching and self-invalidation), slipstream improves performance for applications that have reached their scalability limit.

Additionally, we introduce how slipstream execution mode can provide a confidence mechanism for barrier speculation [44, 17, 32], a technique used to hide barrier synchronization overhead. This optimization ignores the intended dependency imposed by the synchronization and tries to execute code following the synchronization. As this execution involves many memory transactions, there is a chance of dependency violations and thus rollbacks. Frequent rollbacks can not only render the scheme useless, but also can harm performance because of the added traffic/contention to the memory system and the premature migration of data. We introduce a mechanism, based on slipstream mode, with which we predict dependencies and then block/delay speculation when a dependency violation is predicted. This technique reduces

rollbacks and preserves the gains obtained by speculative execution.

We also address the problem of supporting slipstream execution mode for programs written using the OpenMP standard. The compiler can hide the details of slipstream execution, so that programs can transparently use slipstream mode to enhance performance. Specifically, one new directive and one runtime variable are introduced that enable compilation for slipstream mode. The use of slipstream mode and the slipstream synchronization mechanism are selected at runtime, so a single executable image can be used with and without slipstream support. If desired, a programmer can use the directive to specify slipstream parameters for distinct regions of the code.

1.2 Research Objectives

In this work, we investigate the following:

- Exploring the use of CMP for shared memory machines: CMP is intended to provide more opportunities for parallelism by having multiple threads utilizing the functional units. The use for these CMP chips in shared memory parallel computers needs more investigation.

- Investigating a scheme that addresses parallelization overheads: We aim at reducing the overheads of parallel programs by hiding the latency of accessing memory and also by hiding the effects of synchronization.

- Devising future-based predictors for shared memory machines: Future based prediction is recently proposed for a single node system. The potential of highly accurate information provided by these schemes is yet to be explored for shared memory machines.

- Investigating the role of shared *vs.* local data in parallel program writing to achieve simple and accurate future information predictors.

- Enhancing the performance of compiler-based parallelization using hardware optimization: Due to the complexity of parallel programming, compiler assisted

parallelization is gaining wider acceptance. Compiler-assisted parallelization may not, though, provide hardware-specific optimizations. In this work, we aim at enhancing the performance of applications written with OpenMP (a new widely accepted standard for writing shared memory programs) running on CMP-based shared memory system.

- Investigating different mechanisms for identifying last touch for shared memory programs. Identifying last touch is used to timed preemptive coherence actions on cache lines in several optimization, for instance, self-invalidation.

- Providing a confidence mechanism for barrier speculation. This helps to reduce misspeculation and rollbacks that can hurt performance.

1.3 Contributions

The main contributions of this work are

- Introducing a future-based speculation mechanism for shared memory machines composed of CMP chips. The proposed scheme is simple due to our observation of the rule of shared variables *vs.* private variables in shared memory programming.

- Introducing multiple software implementations to support this execution mode. Notably, we extend OpenMP standard to support slipstream execution mode, a new widely accepted standard.

- Illustrating how to use these future information for prefetching, a consumer-initiated communication. We also extend the scheme, using special memory transactions, to support a form of self-invalidation, a producer-initiated communication.

- Proposing and assessing the quality of different identification mechanisms for last touch or write to a cache line. These touch predictors can be used in different optimizations like self-invalidation and speculation after barrier synchronization.

We proposed a last touch predictor that has a reasonable requirement along with very high accuracy.

- Designing a confidence mechanism for speculation after barrier synchronization. This mechanism reduces the chance of rollbacks. We also propose techniques to reduce the speculation cost/overhead.

1.4 Thesis Outline

Chapter 2 of this report reviews related work in latency hiding, synchronization hiding, and precomputation. Chapter 3 introduces the slipstream model for shared memory systems. It discusses how to create future-based information both simply and accurately. Chapter 4 discusses two models for implementing the software part of the slipstream model. Chapter 5 covers the use of slipstream for latency hiding using both prefetching and self-invalidation. In Chapter 6, we introduce and evaluate different identification mechanism for last access or write of a cache line. Chapter 7 illustrates the use of slipstream for synchronization speculation. Chapter 8 provides conclusions and possible future work.

Chapter 2

Related Work

In this chapter, we review approaches adopted to hide or reduce overheads of executing programs. Depending on the amount of observed (targeted) latency, different techniques are used. In this chapter, we aim to span different techniques that target these latencies while giving a special attention to memory latency. Memory latency is a problem that appears for both single processor machines and multiprocessor machines. Scientific workloads represent a significant sector of workloads used on shared memory machines, which are our target research machines. While larger cache hierarchies have proved to be effective in reducing these latencies, it is common for scientific workloads to spend large portion of their run time stalled on memory requests.

2.1 Latency Hiding Techniques

A bottleneck in obtaining a high utilization of CPU cycles is stall time incurred while accessing different units. This latency can be small, e.g., an L1 miss that hits in L2 or a floating point operation. This latency can be hidden using Microarchitecture techniques, such as, out-of-order superscalar. Much longer latencies are incurred to access memory. This memory latency is usually handled using prefetching and/or self-invalidation (for shared memory machines).

In this section, we overview architectural support to hide these different categories of latency.

2.1.1 Microarchitecture Approaches

Instruction-level parallelism can be viewed as a small latency hiding technique, as it tries to overlap the execution of multiple instructions. If the execution of these instructions requires some additional latency, then enough work still exists to keep functional units busy.

Superscalar refers to the microarchitecture design that allows more than one instruction to be executed in the same cycle. These designs may be based on an out-of-order core [49], or in-order core, for example VLIW [15].

An out-of-order microarchitecture tries to increase performance by dynamically scheduling instructions [49]. This architecture may overlap instructions that the compiler cannot overlap as it has the runtime information that is not available to the compiler. On the other hand the scope of the scheduler is usually small (compared with that available to the compiler) and also it increases the hardware complexity.

Very-Long Instruction Word (VLIW) techniques try to extract and to exploit instruction level parallelism at the compiler level [15]. The advantage of doing it at the compiler level (*vs.* hardware level) is the larger scope that the compiler can look at to extract parallelism. The objective of VLIW is to eliminate complicated instruction scheduling and dispatching to achieve a faster, less expensive processor.

Simultaneous multithreading (SMT) [50] multiplexes resources between threads within a cycle. SMT tries to search for independent instructions from independent threads instead of extracting them from the same thread. The performance of an SMT can outperform an ordinary ILP processor if there is a lack of independent instructions in the same thread.

The decoupled access/execute (DAE) architecture [46] decomposes a program into two separate streams. A memory access stream slips ahead of an execution stream, and supplies data to it. DAE relies on finding decoupled instruction streams either at run-time or with the support of a compiler.

Most of the latencies of the functional units and L1 caches are hidden using these techniques. Actually, these approaches, by hiding small latencies, make most of the stall time due to the memory system. They also expose memory latency more by

overlapping busy cycles.

2.1.2 Prefetching

Prefetching is a technique that reduces perceived memory latency by requesting cache lines before they are needed by the program. Prefetching may be guided by hardware prediction tables [10], by the programmer or compiler [30, 35], or by pre-computation [54, 5, 6, 12, 13, 29, 42, 43, 35].

Hardware Prefetching can be from processor side [10] or memory side [47]. In both cases, memory references are monitored to infer future prefetching opportunities. These inferred references may be incorrect, thus causing cache pollution, and consuming more memory bandwidth.

In software controlled prefetching [35, 30], execution time is reduced despite the added instructions for prefetching. The challenge that faces this approach lies in the placement of prefetch instructions in the target applications. This can be complicated, if there is a large variability in memory access time.

Timing is a challenge to do effective prefetching, as data should be requested early enough to hide memory latency. The data should not be requested so early that the data prefetched will be replaced before being used. The required time to prefetch a block is usually variable. In uniprocessor case, this variability is due to memory state (contended or not) and bus bandwidth. In multiprocessor configuration, additional factors may come into play, such as data placement for distributed memory, memory state of the cache line, memory controller availability, network bandwidth, etc. Timing is more critical in the multiprocessor configuration- not only might prefetched block suffer replacement, it also may be invalidated by another process. This extra invalidation is an overhead that may hurt the performance of the other processes.

2.1.3 Coherence Optimization

Coherence optimization techniques aim at reducing or overlapping transactions involved to migrate data in cache-coherent shared memory systems. The basic idea is

to try to predict the next coherence action based on the current signature of coherence actions on a certain cache block. History-based techniques are used to correlate stable coherence patterns to future actions.

These predictors can be at the node or at the directory. They can also be address-based or PC-based. An example of node-based optimizations is proposed by Kaxiras [21]. In this optimization, a node that issues a non-atomic read-write sequence to the same cache block can request exclusive ownership of the line the first time it sees the read. This potentially reduces the communication overhead and also reduces the chance of stalling the processor. Mukherjee [36] proposes monitoring coherence actions at the directory and trying to correlate this with future actions. They found that cache blocks exhibits a predictable pattern of coherence actions. Mukherjee [36] reported the accuracy of a his proposed predictor but not the performance improvement if it is used. Lai et al. [24] use a similar prediction technique and shows performance improvement due to the use of these predictors.

2.1.4 Self-invalidation

Self-invalidation (SI) [27, 25] is a technique to reduce the latency of coherence misses. SI advises a processor to invalidate its local copy of a cache line before a conflicting access occurs. When successful, this will reduce invalidation messages and writeback requests. A subsequent load from another processor will find the data in memory, rather than having to request a writeback from the owning cache. A subsequent store will acquire an exclusive copy from memory without having to invalidate copies in other caches.

Dynamic self-invalidation (DSI) [27] describes two methods for identifying lines for self-invalidation. One uses extra coherence states and the other uses version numbers to remember past conflicts. Past conflicts are used to infer future conflicts. Lines are self-invalidated at synchronization points.

Last-touch prediction [25] improves on SI by more precisely identifying the last touch to a cache line, so that self-invalidation is done as early as possible. This also reduces self-invalidation bursts at synchronization points. However, history-based

last-touch prediction may require large hardware prediction tables because they are indexed by line address and must accommodate large working set sizes.

2.2 Synchronization Overhead Reduction/Hiding Techniques

Synchronization is one of the bottlenecks for parallel program scalability. The contribution of synchronization to total execution time tends to increase as the number of processors increases, especially if problem size remains fixed. Two major synchronization constructs grasp the attention of researchers. The first one is locks and the second is barriers.

Synchronization primitives usually reflect some dependency constraints and sharing pattern characteristic. For example, locks are best used to synchronize access to shared memory locations that are migratory. Locks guarantee mutual exclusion and organize handing off during migration. The order of execution and migration is not strict. Dependency is not strict in the sense that there is no necessary global order of execution. The outcome exhibits an arbitrary execution order.

Barriers, on the other hand, are a bulk synchronization primitive. They synchronize many producer-consumer relations, effectively taking the place of many point-to-point synchronization events. The reason for using them is the ease of orchestrating programs. Barriers are also appropriate when a large data structure must be synchronized among multiple processes.

Though convenient for programming, the use of a barrier is a source of inefficiency in parallel programs. This inefficiency generally gets worse when we increase the degree of parallelism. Replacing a barrier with point to point synchronization is not only hard to implement but may not be also an efficient solution. The reason is that we may need synchronization variables proportional to the data set. Another problem is that we will need to check a synchronization variable before consuming each variable, which can be a time-consuming process.

2.2.1 Critical Sections

Protecting critical sections using lock synchronization is a common practice in shared memory programs. The data protected in a critical section can be single variable or a large data structure like hash tables. If the data accessed is large or not written during the critical sections, then this critical section could have been concurrently executed by multiple processes.

When there is a low chance of collision of accesses (writes to the same cache line), it is beneficial not to serialize access to data. Instead, it is advisable to use atomic (lock-free) algorithms [51, 34]. If the collision probability is high (for example, same variables are accessed and are always written), then it is better to serialize using critical sections.

Rajwar et al. [40] advocate making the choice between atomic execution and serialization at the hardware level. Their point is that parallel programming difficulty makes programmer reasoning uneasy job. Martinez et al. [32] try to reduce the impact of serialization in critical sections by executing them speculatively in optimistic fashion (assuming no collision). Both techniques use the coherence protocol to detect violations. Violations trigger rollbacks and retries. Martinez et al. proposal does not provide dynamic switching to non-speculative mode if multiple violations occurs.

2.2.2 Barrier Synchronization

Barriers often contribute the most to synchronization time, especially for scientific computation benchmarks like SPLASH-2 [52]. The increase in barrier synchronization time is due to workload imbalance and the communication overhead associated with the release of the barrier. The fluctuation of arrival times for a large number of customers to the barrier results in increasing the average waiting time. Asynchronous operations, such as memory references and I/O operations, may also result in artificial workload imbalance.

Barrier latency can be hidden if useful work is introduced during the waiting time [14, 44]. One way of doing this is to try speculation after barrier synchronization, introduced by Sato et al. [44]. The speculation is based on consuming the current

value of a variable, speculating that it has already been produced. The success of speculation is due to the chance of having the values that are needed for loads to be already produced. If a production by the non-speculative process comes after consumption by the speculative process, the speculation fails and rollback is necessary. The penalty in the case of rollback can be higher than the benefit of speculation. This proposal provided a coherence protocol extension on bus-based symmetric multiprocessor system.

Martinez et al. [32] use a similar optimistic approach in executing instructions after barrier synchronization. This proposal uses unmodified cache coherence protocol and propose an implementation on CC-NUMA system. It also provides a mechanism for speculation inside critical sections.

2.3 Future Information Based Speculation

Recently, program-based future information are proposed to assist speculative execution. This future information tackles both ILP bottlenecks and memory latency problem.

2.3.1 Slipstream Paradigm

Slipstream paradigm [48] uses redundant execution to speed up sequential programs. A slipstream processor runs two redundant copies of a program on a dual-processor CMP. A significant number of dynamic instructions are speculatively removed from one of the program copies (called the A-stream) , without sacrificing its ability to make correct forward progress. The unreduced program (called the *R-stream*) R-stream exploits the A-stream outcomes to get very accurate branch and value predictions. The predictions are more accurate than predictions made by conventional history-based predictors because they are produced by future program computation. The speculative A-stream occasionally (but infrequently) goes astray. The R-stream serves as a checker of the speculative A-stream and redirects it when needed.

2.3.2 Pre-execution/Pre-computation Paradigms

Pre-computation uses helper threads to compute the addresses of problem loads (loads that frequently miss in the cache) ahead of the full program [35, 13, 42, 54, 5, 6, 12, 29, 43]. Problem loads are explicitly identified and targeted, through profiling [43, 54] or dynamic identification mechanisms [42, 11]. Then, the computation slices that generate the addresses of problem loads are extracted from the full program either manually [54], by the compiler [29], or by hardware mechanisms [11]. Finally, microarchitectural threads are forked as needed to remove long latency operations from the critical path of the program, paying special attention to timely forking [11]. A different flavor of helper thread exploits intelligent memory architectures that have processors embedded in main memory [47]. A user-level thread implements memory-side prefetch algorithms in software. Prefetch predictions are stored in memory instead of in custom hardware tables, the tables are software-managed, and the prefetch algorithm is flexible.

Helper threads operate in the context of sequential programs. They are effective for small to medium latencies. Targeting long latencies like memory, especially as the gap is increasing in speed with CPU, poses challenges to these techniques. These forms of prefetching also specialize in moving data between a single processor and memory hierarchy. They are not tailored to coordinating communication among distributed processor caches.

Chapter 3

Slipstream for Shared Memory Systems

Slipstream for shared memory system tries to use a compute-centric design in communicate-centric fashion. In this mode, we use one processor of the CMP chip to produce memory references that may benefit the other processor on the CMP chip. These memory references can be used to enable multiple optimizations as will be discussed in later chapters. The usefulness of these informations is affected by two axes: functional accuracy and timing accuracy. In this chapter, we address how to produce functionally accurate information for shared memory programs based on the observation of the roles of shared variables and private variables in program writing. We also present how to control the timing of using these information using a simple synchronization model.

An example of supercomputer project that adopts a similar concept is the Blue-Gene/L project [4] at IBM. In this project, each node is composed of dual-processor CMP chips. One processor of each node may be used for computation while the other is used for communication. The programming model supported is message passing. The machine also supports having both processors to work for computation if the computation-to-communication ratio is large.

```
myid = get_id();
nprocs = get_totalprocs();
mymin = 1+myid * n / nprocs;
mymax = mymin + n / nprocs - 1;

for(i=mymin;i<mymax;i++)
        for(j=1;j<n-1;j++)
                Y[i][j] = (X[i-1][j] + X[i+1][j]
                        + X[i][j-1] + X[i][j+1])/4;

BARRIER(nprocs);

for(i=mymin;i<mymax;i++)
        for(j=1;j<n-1;j++)
                X[i][j] = (Y[i-1][j] + Y[i+1][j]
                        + Y[i][j-1] + Y[i][j+1])/4;

BARRIER(nprocs);
```

Figure 3.1: Portion of a Red-Black SOR kernel. **X**,**Y** are shared variables. Control flow and address generation are only affected by private variables.

3.1 Shared Variables vs. Local Variables

Figure 3.1 shows a red-black implementation of an SOR (successive over-relaxation) kernel. Shared variables are indicated by bold capital letters. Loops are controlled by local variables, and their limits are a function of the task ID. Ignoring the stores to shared variables **(X** and **Y)** leads to incorrect algorithm results, but does not affect the control flow or the information about which memory locations are accessed or the types of the accesses. Because the A-stream and the R-stream have the same task ID, *myid* in our example, then they are expected to perform the same operations on the same portion of shared data.

3.2 CMP Based Shared Memory System

In this section, we investigate how to create a future-based prediction scheme based on the observation discussed in Section 3.1.

3.2.1 Forming Shorter Stream

Slipstream execution mode requires forming a shorter A-stream task for predictive purposes. An effective approach to reducing an A-stream is to remove its long-latency communication events. In shared memory multiprocessors, these are synchronizations (barriers, locks, and events) and accesses to shared memory (loads and stores). Two of these, synchronization and shared memory stores, can be skipped for many programs without affecting the control flow or the A-stream's ability to predict access patterns for shared data.

In order to skip synchronizations, the system-provided routines for barriers, locks, and events (for example, the ANL macros [31] used by the Splash-2 benchmarks) are modified to support tasks running in slipstream mode. The A-stream tasks do not perform the synchronization routine, but the R-stream tasks execute them normally.

Synchronization is used to define dependencies in accessing shared variables. Skipping these routines makes the A-stream speculative, since we cannot guarantee that the dependencies imposed by the synchronization will be met. We cannot allow values produced by the A-stream to be stored to shared memory, unless we have the ability to roll back that shared state to its previous value in case of a misspeculation. Our solution is to have the A-stream processor discard stores to shared memory. (The store instruction is executed in the processor pipeline, but it is not committed. Although the store is not committed, it may be converted into a prefetch-exclusive request, as discussed in Chapter 5.) The R-streams throughout the system are not corrupted by erroneous A-stream values, because local changes to shared variables are never stored and never made visible to the other tasks. This also shortens the A-stream further, since shared memory stores might otherwise incur long latencies due to invalidation requests and acknowledgments.

The shared data loaded by the A-stream may be incorrect, if the load occurs before the producing task has stored the final value. In the original program, this dependency is enforced through synchronization. Even though the A-stream brings this data into the shared L2 cache prematurely, it will not affect the R-stream's correctness, because the R-stream will observe the synchronization before consuming

the data. If the producer changes the data before the synchronization, the copy loaded by the A-stream is invalidated and the R-stream will retrieve the correct value when needed. The A-stream prefetch is ineffective, from the R-stream's point of view, but it does allow the A-stream to make progress and stay ahead of the R-stream.

Since the role of the A-stream is to collect access information and to prefetch shared data, we must be confident that using speculative data does not significantly affect control flow or address generation. Otherwise, the access patterns predicted by the A-stream will be inaccurate, and we may add to the memory traffic of the system by loading values that are unnecessary.

Fortunately, many parallel programs are written in the SPMD (Single Program, Multiple Data) style, in which each task executes the same code but accesses different portions of the shared data. These programs rely mostly on local variables for address computation and control flow determination. These programs use a unique task ID to identify the portion of the shared data accessed by each task. Barriers identify phases of execution and guarantee no inter-phase dependency violations. Parallel scientific numerical algorithms are good examples of this class of applications.

We have identified only three types of shared variables that typically affect control flow or address generation:

- ***Synchronization variables*** affect control flow, because they grant or deny access to regions of code. Variables that are referenced inside system-provided synchronization routines are no problem, because those routines are modified to skip references to the variables for the A-stream task. A user-defined synchronization variable, such as a simple flag variable, may cause a local divergence in control flow, because the A-stream might wait on the variable, while the R-stream might not. This divergence is only temporary, however. The A-stream may be prevented from moving far ahead of the R-stream, but the accuracy of its data accesses is not likely to be affected.

- ***Reduction variables***, which are used to compute global values, such as the minimum or maximum of a set of data, may affect control flow. For example, a comparison between the task's local minimum and the global (shared) minimum

determines whether the task should perform a store to the global minimum. The effect on control flow is localized and should not cause a divergence of the A-stream and R-stream. The accuracy of the information collected by the A-stream may be degraded, however.

- ***Dynamic scheduling*** relies on shared information to make a decision about which task or sub-task to execute next. These decisions are time-dependent, so it is likely that the A-stream would make a different decision, and access different data, than the R-stream that comes later. Dynamic scheduling can be accommodated in slipstream mode. If the scheduling code is identified, the A-stream may skip the code and wait for the R-stream to catch up (using the A-R synchronization mechanisms described below). Once a scheduling decision is made by the R-stream, the A-stream will again run ahead and collect data access information. The Barnes-Hut benchmark from Splash-2 is an example of a program that does its own dynamic scheduling. In that case, programmer intervention is needed to reduce the likelihood of A-stream divergence. However, if dynamic scheduling is implemented by a compiler, as in the case of OpenMP [2] programs, then support for slipstream mode is transparent to the programmer.

These "problem" variables represent a small fraction of accesses. We will consider dynamic scheduling in depth in Chapter 5. All optimization that will be described on the following sections are for working set shared data (which do not belong to the above three classes). We may refer for conciseness to these working set shared data as shared data.

3.2.2 L1/L2 Coherence for Slipstream

Since shared data generally does not affect control flow and address generation, A-stream consumption of stale shared data does not affect its accuracy for collecting shared memory access behavior. On a CMP, it is usually required to maintain L1 caches coherent with the L2 cache. For working set shared data, this condition can

be relaxed by allowing the A-stream to consume non-coherent shared data. This relaxation helps alleviate the effect of the A-stream contending with R-stream on the shared data in the L2. The A-stream in this case can consume stale data from L1. We do maintain inclusion, so that A-stream prefetches data if does not exist in the L2 cache.

To maintain this semantic, we differentiate between an L1 invalidation for coherency with the other L1 cache and an invalidation due to an external event or replacement. For invalidation due to a write by the other L1 cache, the A-stream L1 cache controller keeps a speculative copy of the cache line. External invalidation or replacement will do an invalidation for both L1 caches. In this case, the line should be replaced from the L1 cache.

3.2.3 Stream Synchronization

Synchronization between an R-stream and its corresponding A-stream is required for two reasons. First, we must correct an A-stream that has taken a completely wrong control path and is generating useless data access predictions. Second, we want to limit how far the A-stream gets ahead, so that its prefetches are not issued so early that the prefetched lines are replaced or invalidated in the L2 cache before the R-stream uses them.

We couple the A-R synchronization mechanism to the barrier and event-wait synchronizations specified in the program. These events typically represent transitions between phases of computation, so they are natural points at which to manage the interaction between the two streams. Furthermore, the library routines that implement these synchronization constructs already require modification in order to allow the A-stream to skip them. Now we modify them a bit further: when the A-stream reaches a barrier or event-wait, it either skips the synchronization or it waits for its local R-stream to give permission to continue. We define a *session* as a sequence of instructions that ends with a barrier or event-wait synchronization. One of the parameters that will be controlled by the A-R synchronization mechanism is the number of sessions that the A-stream is allowed to run ahead of the R-stream.

We require a single semaphore between each A-stream/R-stream pair to control their synchronization. For our experiments, we have assumed a shared hardware register, but any shared location that supports an atomic read-modify-write operation is sufficient. Using this semaphore, we control (a) how many synchronization events the A-stream can skip without waiting for the R-stream, and (b) whether the synchronization is local (involving only the companion R-stream) or global (involving all R-streams).

The initial value of the semaphore indicates how many sessions the A-stream may proceed ahead of the R-stream. As depicted in Figure 3.2, this can be viewed as creating an initial pool of tokens that are consumed as the A-stream enters a new session. When there are no tokens, the A-stream may not proceed. The R-stream issues tokens by incrementing the semaphore counter.

Two different types of synchronization – *local* and *global* – are enabled by controlling when the R-stream inserts a new token. If the R-stream inserts a token as it *enters* the barrier or wait routine, then the continued progress of the A-stream depends only on its local R-stream. If the R-stream inserts a token as it *leaves* the barrier or wait routine, then the continued progress of the A-stream depends on *all* of the R-streams participating in the synchronization. (In the case of a barrier, this is all of the R-streams waiting on the barrier. In the case of an event-wait, this is the R-stream that signals the event.)

To reduce the overhead of synchronization, we assume shared registers between the processors sharing a CMP. This register can be memory addressable. The semantics of the operations required on these registers are as follows:

INIT_SEMA(sema_id,init_val). This operation is used to initialize a shared register (identified by *sema_id*) to an initial value (*init_val*). This operation is simply implemented by moving an immediate value to a memory addressable register. This initialization must be done for each CMP running in slipstream mode.

PSEMA(sema_id). This operation is used to decrement *sema_id* semaphore. If the result is less than zero, the calling process stalls until condition is changed. The

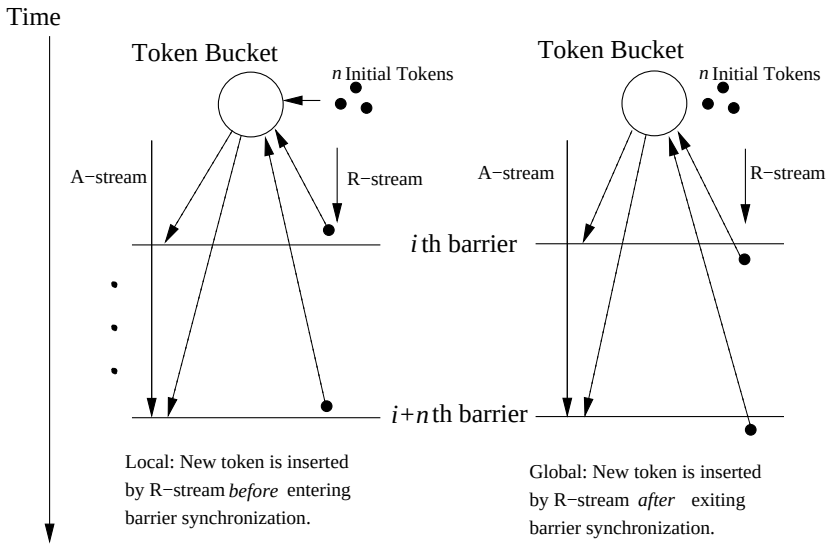


Figure 3.2: Slipstream Synchronization Model; A-stream allowable progress is controlled by the number of tokens and the insertion type.

processor should be able to execute other system routines and handle interrupts. This operation is done only by the A-stream. This operation can also trigger releasing the stall of an R-stream, in case it was waiting on condition that is satisfied by the latest state of the register.

VSEMA(sema_id). This operation is used to increment *sema_id* semaphore. The new state of the register may remove the stall of an A-stream if the value of the semaphore becomes greater or equal to zero.

WAIT(sema_id,val). This operation checks the state of semaphore *sema_id* against *val*. If this is issued by an A-stream, then the check is if state[*sema_id*]<val. If this is issued by an R-stream, then the check is if state[*sema_id*]>=val. If the condition is satisfied, then calling stream is stalled; otherwise, it proceeds normally. As mentioned earlier, this stall should not prevent the processor from executing other processes or to service interrupts.

If A-stream and R-stream are loosely scheduled by the Operating System, then we should allow no more than one A-stream/R-stream pair to use the CMP. This is not an additional restriction, as deciding to have multiple jobs negates the need to use

slipstream mode in the first place.

To enable the use of slipstream mode, the programmer selects slipstream-aware parallel libraries that control task creation, synchronization, and so forth. At run time, if the application user chooses slipstream mode, then the slipstream library routine creates two copies of each task, and assigns one copy to each of the processors on a CMP. Just as in double mode (with separate parallel tasks), each task has its own private data, but shared data are not replicated.

3.2.4 Deviation Detection/Recovery

The A-R synchronization points are also used to check for a deviating A-stream – that is, one that has taken a significantly different control path than the correct path, represented by the R-stream. The checking is very simple: if the R-stream reaches the end of a session before the A-stream, we assume the A-stream has deviated. The R-stream checks the number of tokens available against the initial token count. If the number of tokens equals the initial count, then the R-stream predict that the A-stream has not visited this synchronization. This check does not include any notion of whether the data access predictions from the A-stream have been accurate or not.

The recovery mechanism is equally simple: the R-stream kills the A-stream task and forks a new one. This may be expensive, depending on the task creation model. In our experience, however, the benchmarks used do not require recovery, as they do not diverge.

3.2.5 Slipstream I/O

There is one other need for synchronization between the A-stream and R-stream. Some global operations, such as system calls, I/O, and shared memory allocations, must only be performed once, since they impact global system state. Except for input operations, the A-stream skips these operations. For input, the A-stream synchronizes using a local semaphore, similar to the one described above. After the operation is completed by the R-stream, its return value is passed to the A-stream through a shared memory location. This implies the need for a slipstream-aware system library.

Chapter 4

Slipstream Software Support

In this chapter, we introduce the support needed for slipstream from the software side. We investigate two models representing two methods of writing shared memory programs.

The first model is based on macros that provide simple primitives for parallelization. To be efficiently utilized, these macros require an experienced programmer of shared memory programming. They also impose some challenge to porting applications transparently to slipstream mode.

The second model represents compiler-assisted parallelization directives. This paradigm provides easy to use directives to instrument how a serial program can be parallelized. Most of the parallelization process is done by the compiler. This model relaxes some of the limitations imposed for porting applications to slipstream mode. It allows different scheduling techniques. It also facilitates changing the type of slipstream A-R synchronization for different phases of program execution.

4.1 ANL Macros

ANL [31] macros aim at providing a portable interface for writing parallel programs. In this section, we overview necessary modifications for the ANL macros related to shared memory programming [19].

CREATE. This macro creates a slave process that is to carry out a specific job.

In slipstream mode, two processes are created, A-stream and R-stream. This pair of processes must share the same CMP. We use the IRIX system call sysmp(MP_MUSTRUN, proc_id) to force the necessary affinity for slipstream.

BARRIER. This function, besides synchronizing R-streams, serves to synchronize A-stream with R-stream. They also serve as deviation checking and recovery points. A-stream does not exit these function unless at least one token exist in the synchronization bucket. The R-stream also checks if the A-stream has entered this routine before it by checking the number of tokens in the bucket compared with the initial value. If the R-stream detects a deviation, it may kill and refork another A-stream. In our implementation, R-stream waits for small number of cycles before invoking the recovery routine.

WAIT_PAUSE. This macro serves as a second synchronization point between both streams. This has the same mechanism implemented in BARRIER.

CLEAR_PAUSE, SET_PAUSE, **LOCK**, and **UNLOCK**. Only the R-stream should execute these macros. A-stream skips them.

MAIN_ENV. This macro carries out all the declarations in addition to slipstream special declarations.

INIT_ENV. This macro handles the runtime information about the desired execution mode (slipstream or otherwise). It also passes information about the synchronization method to the hardware. The shared memory address space is reserved in this macro also. The portion of shared memory space that requires special handling is conveyed to the hardware.

G_MALLOC. This macro allocates shared data. To simplify the hardware support, shared data is not overlapped with local data. If this function is invoked in the parallel region, the A-stream should synchronize with its R-stream to receive the same return of this function.

G_FREE. This macro is used to deallocate shared memory. Only R-stream should execute this function. A-stream skips it.

WAIT_FOR_END. This macro is used by the master to wait for all slaves to finish their assigned job. The master should not wait for A-stream completions.

GET_PID. Both A-stream and R-stream should be returned the same process id.

LOCKINIT, **ALOCKINIT**, **SLINIT**, and **SEMINIT**. These macros initialize synchronization variables and should be executed by R-stream only.

The following macros are introduced to provide extended functionality for slipstream explicit programming.

AR_BARRIER. This macro translates to a barrier routine that should be respected not only by R-streams, but also by A-streams. We used this macro with SPLASH-2 Ocean to prevent deviation of A-stream in multi.C functions as shown in Figure 4.1. The computed error value controls how the algorithms continue. Thus, this shared variable affects control flow and A-stream should be forced to synchronize with its R-stream (no matter what is the global synchronization method) to make the correct decision about how the algorithms continue.

IS_A_STREAM. This macro can be used to check if the code is to be executed by A-stream.

IS_R_STREAM. This macro can be use to check if the code is to be executed by R-stream. Along with the previous macros, programmer can explicitly specify portions of the code that need not be executed by A-stream. If the programming involves an I/O library that is not slipstream-aware, then it is advisable to protect these portions of the code with if(IS_R_STREAM).

AR_SYNC. This macro is used to synchronize A-stream and R-stream so that the A-stream will not consume data not already produced. This macro can be used in dynamic scheduling code and also in some system routines. In our implementation, we use this synchronization to implement G_MALLOC.

```
LOCK(locks->error_lock)
      if(local_err > multi->err_multi)
      {
            multi->err_multi = local_err;
      }
UNLOCK(locks->error_lock)
AR_BARRIER(bars->error_barrier,nprocs)
g_error = multi->err_multi;
```

Figure 4.1: Using AR_BARRIER in OCEAN multigrid.

The four extension macros described above (except for AR_BARRIER) were not explicitly used in porting applications used in this study. The macro IS_A_STREAM is inserted to protect execution of some of the I/O functions, for instance printf.

4.1.1 Limitations

ANL macros provide simple primitives for parallel programming. This leaves the space open for the programmer's imagination to develop shared memory programs. The flexibility provided by ANL makes it harder to program and difficult to port all applications transparently to slipstream mode.

The limitations that face transparent porting of SPLASH-2 applications to slipstream are:

- Difficulty for supporting programs with embedded dynamic scheduling. Barnes-Hut is an example of this class of applications.
- Identification of shared variables under some models may be difficult.
- Forcing the use of a single synchronization mode for the whole program.

4.2 OpenMP Support

In this section, we discuss how to extend OpenMP to support slipstream [18]. OpenMP represents an easy standard that enables incremental compiler-assisted par-

allelization. The parallelization automation provided by this standard enables alleviation of the limitations imposed on using slipstream with ANL macros.

4.2.1 OpenMP Overview

OpenMP [2] is an emerging directive-based standard for shared-memory parallel programming. It allows simple incremental parallelization for applications. OpenMP does not provide facilities to control data locality or coherence, as these features are platform dependent. Portability of OpenMP application puts the burden on the compiler and hardware to achieve decent performance.

While a compiler can do analysis to remove unnecessary synchronization and to optimize for locality of data accesses, the overhead of parallelization *vs.* performance gain cannot always be determined at compile time. For example, we cannot determine if parallelizing a certain loop will be worthwhile without knowing the loop iteration count, which can be a runtime variable. Most data-parallel applications use the same code to solve different problem sizes. Likewise the upper limit of parallelization for decent performance is dependent on the runtime information like problem size and the underlying architecture. This encouraged the OpenMP standard to include environment variables to facilitate changing scheduling and the degree of parallelism at runtime.

Other researchers have proposed extensions to OpenMP to express architecture specific optimizations, such as data distribution directives for CC-NUMA [7] and software-DSM [33] systems. Such extensions may inhibit portability, but they can be ignored by systems for which they do not apply. They give the programmer another tool for tuning performance without explicitly modifying the application program.

OpenMP standard provides a tool to parallelize program in an incremental way. Figure 4.2 shows a fragment of a Laplace equation solver. The code is essentially a serial code augmented by pragma directives that control how this program is to be parallelized. This fragment includes some of the most commonly used pragmas in the OpenMP standard.

Pragma *parallel* tells the compiler that a parallel region is to be encountered; this

may correspond in one implementation to a fork for a number of processes.

Pragma *for* parallelizes the loop immediately following it. The number of iterations is assigned to the available processes, based on a certain scheduling policy. This pragma implies a barrier at the end of it unless *a nowait* statement is specified.

Pragma *critical* serializes the access to a section of code to guarantee mutual exclusion. Pragma *barrier* is an explicit barrier synchronization. Pragma *master* protects a section from being executed except by the master. Pragma *single* causes a section to be executed by only one process (usually the first to hit the section).

OpenMP also provides facilities for functional parallelism called *sections*. Each thread of execution is expected to execute one of these sections. These sections usually impose an upper limit of the degree of parallelism.

Other facilities are provided through pragma and/or runtime library calls to support reduction operations, control scheduling, synchronize, and get/set thread information. To maintain portability, the OpenMP standard avoids having any hardware-specific pragma.

4.2.2 Requirements for Slipstream Mode

Slipstream support for OpenMP is mostly done through modifying the underlying library that manages threads of execution. The changes to the library are summarized as follows:

- Process creation: The compiler creates a pool of slaves at the start of running the program (probably equal to the number of processors). Slipstream mode is assumed to execute on a system with dual-processor CMP nodes. We assume that resources are allocated in CMP-based units. Following this resource dedication, the number of processes must be even.

- Shared address space: To simplify the support for slipstream, the virtual shared space must be either contiguous or non-contiguous but not interleaved with private space, to ease delineation of what is shared and what is not shared. In OpenMP, there are explicit semantics controlling which variables are shared

```
double u[XSIZE+2][YSIZE+2],uu[XSIZE+2][YSIZE+2];

laplace_solver()
{
    int x,y,k;
    double sum;
#pragma omp parallel private(k,x,y)
{
    for(k = 0; k < NITER; k++){
        /* old <- new */
#pragma omp for
        for(x = 1; x <= XSIZE; x++)
          for(y = 1; y <= YSIZE; y++)
            uu[x][y] = u[x][y];
        /* update */
#pragma omp for
        for(x = 1; x <= XSIZE; x++)
          for(y = 1; y <= YSIZE; y++)
            u[x][y] = (uu[x-1][y] + uu[x+1][y]
                    + uu[x][y-1] + uu[x][y+1])/4.0;
    }
}
    /* check sum */
    sum = 0.0;
#pragma omp parallel for private(y) reduction(+:sum)
    for(x = 1; x <= XSIZE; x++)
        for(y = 1; y <= YSIZE; y++)
          sum += (uu[x][y]-u[x][y]);

    printf("sum = %g\n",sum);
}
```

Figure 4.2: OpenMP Laplace solver

and which are private. The underlying thread model decides how to specify the shared *vs.* private virtual space. For UNIX processes, for example, it is a common practice to allocate shared virtual addresses in a contiguous space. For POSIX threads, on the other hand, shared space is not necessarily contiguous. In this case, it is the compiler's job to guarantee that shared space is not interleaved with private space.

- Deciding execution mode: Slipstream is considered additive to normal mode. We assume that the same binary should run for both normal and slipstream mode. An application (or a kernel) declares its intent to run in slipstream mode by writing to a control register. By default, slipstream mode is deactivated. The application can have an argument to declare the user's intent to activate slipstream mode.

- Stream synchronization: As discussed earlier, synchronization points have an impact on performance, as they control how far ahead we allow the A-stream to execute. Possible A-stream divergence is also checked during synchronization, and recovery is invoked if divergence is detected. In OpenMP, suitable points for this kind synchronization are the runtime barriers that are inserted due to parallelization constructs. Barriers used to do internal management by the library should not be skipped by A-streams and thus do not require stream synchronization/recovery.

- I/O operations: I/O operations cause irreversible effect on the system. These operations should not be executed by the (speculative) A-stream. *Input* operations may require synchronizations between the A-stream and R-stream, as the A-stream should see the same image of the data that the R-stream sees. *Output* operations do not requires this kind of synchronization. The additional synchronization for input operations does not pose a practical problem as they are usually done in the serial part of the code (that is executed by only one thread, e.g., the master). Additionally, these operations are usually very slow, which makes stalling the A-stream a good idea—being very far ahead could hurt

performance in this case.

- Explicit library calls

  The following library calls are modified to reflect the current execution mode. We assume that execution mode (including AR-synchronization, if slipstream is activated) cannot be changed within a parallel construct. So, once this execution mode of a parallel region is established, it remains fixed to the end of this region. Inheritance of execution mode from a parallel region to a nested region within is implementation dependent. The default execution mode may also be implementation dependent.

  1. Reduction: Reduction code can be executed as user code by the A-stream. Special consideration to the task count and and the task id should be taken into account. The A-stream may need to synchronize with its R-stream, if the outcome of the reduction operation will affect program control flow. For example, reduction can be used to compute a global error that affects the iteration or termination condition. This requires adding special types of reduction operations. In practice, termination and continuation are usually decided by the master thread and not in a parallel region, which alleviates the need to synchronize after a reduction operation.
  2. Thread count/id: Thread count and id APIs would depend on the execution mode within a parallel session. If running in slipstream mode, the same id should be returned to processes sharing a CMP. The thread count used by internal library should be half of the total available. While thread id is usually acquired within parallel regions, thread count may be acquired only once on the serial part. This practice (which restricts the OpenMP capability of running with a dynamic number of threads) may force using one mode of execution for all parallel regions. To alleviate this problem, the thread count should simply be acquired in each parallel region. Thread count and id are usually needed if an explicit parallelization is programmed. Actually, the common use scenario does not acquire or

even rely on thread count/id information.

- Parallel constructs

    1. Single: Single section is executed by a single thread in the team (usually the first thread that enters this section). There is no clear way an A-stream can tell that its R-stream will execute this section, as it depends on the order in which R-streams reach this region. Executing by an A-stream that is not paired with the proper R-stream will cause unnecessary migration of data. Obviously, the chance of being lucky gets smaller as the number of CMPs increases. That is why these sections should be skipped by the A-stream.
    2. Master: Unlike single, the R-stream to execute this section is predetermined *a priori*. The A-stream associated with the master can execute it.
    3. DO/For: The execution for this construct is dependent on the scheduling methodology, as will be discussed in detail later.
    4. Atomic: Atomic construct serializes access to data within this construct when there is a small chance for collision of accesses. It is advisable to execute this section by the A-stream, as the data prefetched by the A-stream are highly likely not to be migrated. It is still the programmer's responsibility to use this construct when advisable. Atomic construct can be used to protect a critical section, but performance, as well as optimizations based on common practice, may suffer.
    5. Critical Section: Critical sections guarantee serialization of access to data when there is a high probability for collision. Unless dynamic self-invalidation is supported in slipstream mode, it may be advisable for A-streams to skip critical sections, as they may cause unnecessary migration of data.
    6. Sections: The section construct implements functional parallelism. An A-stream can execute these sections ahead of its R-stream if the scheduler

has a static assignment policy. If dynamic policy is adopted, then the start of this section implies a synchronization between R-stream and A-stream.

7. Flush Directive: This directive aims at synchronizing shared variables and controlling their visibility. It is implied in many other constructs. For hardware cache-coherent systems, this construct maps to void, since the flush semantics are maintained with every transaction to the memory. This directive should be skipped by the A-stream, since it does not produce any shared variables, and thus should not affect the visibility.

4.2.3 Scheduling Strategies

Scheduling technique has an effect on the synchronization between A-stream and R-stream. Static scheduling provides the least restrictive model for slipstream. Other scheduling techniques impose additional synchronization points that may be viewed as additional restrictions or as useful synchronization points that allow more control of how far an A-stream should be ahead of its R-stream. This section is devoted to the interaction between A-stream and R-stream synchronization and scheduling techniques.

4.2.3.1 Static Scheduling

With static scheduling, each thread of execution independently determines the portion of the data that should be manipulated by this thread. To compute this, each thread needs to know the number of threads involved and the amount of work to be done. By adjusting number of threads (to half the available threads), and giving the A-stream and R-stream the same id, each thread can reach the same decision about the task to execute independently. Scheduling under this model does not involve any additional synchronization between R-stream and A-stream. The task assignment for certain loops is done only once at the beginning. This simple scheduling strategy has a very low overhead. It also allows slipstream to execute with different synchronization methods between the A-stream and R-stream. Specifically, the A-stream can be more aggressive (can be more than one session ahead) as its tasks can be computed

independently. Some of the optimizations enabled by the A-stream are tied with certain synchronization models. For example, slipstream self-invalidation is enabled when synchronization model is one-token global.

4.2.3.2 Dynamic Scheduling and Guided Scheduling

In dynamic and guided scheduling, the scheduler tries to optimize load balancing by assigning jobs based on the progress achieved by threads, the amount of work available, and the number of threads involved in solving the problem. Job assignment for the same loop may happen more than once. The programmer may specify a start block size for assignment to processes.

This decision of scheduling parameters (e.g., chunk size of loop iterations) can be problematic by itself. It may require having both a certain problem size and the number of slaves in mind. This scheduling also does not respect cache affinity. Consider an application that has repetitive iterations: there is no guarantee under dynamic scheduling that the same thread will be assigned the same data across iterations. Static scheduling would have given, in this case, the same data to the same process (and same processor assuming no process migration). A proposed affinity scheduling extension [37] attempts to achieve the same result for dynamic scheduling. Cache affinity is not a problem for embarrassingly parallel applications. For this class of application, dynamic scheduling is apparently advantageous, especially if the same amount of data requires a significantly varying execution time. Finally, dynamic scheduling involves additional synchronization overhead, as the scheduling decision must be serialized using a critical section. This serialization is, in fact, a source of load imbalance.

To support slipstream mode with these scheduling techniques, the A-stream needs to know the task assigned to its R-stream. As the task assigned to R-stream depends on the time it asks for the job, the A-stream cannot independently decide it *a priori*. The solution to this problem is that when the A-stream hits a scheduling region, it synchronizes (using syscall hardware semaphore), waiting for its R-stream to reach this region. After the R-stream reaches the scheduling region and gets its scheduling

decision, it declares this scheduling decision by writing it to a shared variable and then releases its A-stream by adding a token to the synchronization semaphore. The A-stream can then acquire the scheduling decision made by its R-stream and can start the work. Although this makes the A-stream lag its R-stream a little bit at the beginning, this is not expected to continue except for few cycles as the data communicated between both streams are cached in the L2 cache and the semaphore used for synchronization is a shared hardware register. Clearly, it is advisable to have a big enough amount of work, not only to allow the A-stream to get ahead of its R-stream, but also to reduce the impact of dynamic scheduling overheads.

This scheduling technique implies a more restrictive synchronization than zero-token global. This does not allow slipstream mode to work except for prefetching. The scheduling decision works as an additional synchronization point between both streams. This may disallow other slipstream optimizations, such as self-invalidation. This behavior can be desirable if the amount of work between two barriers is very large compared with the available cache, which can make the A-stream prefetches evicted before being used or cause replacement of data that is currently in use by its R-stream. These additional synchronization points can help reduce premature prefetches and reduce the frequency of evicting data before being referenced.

4.2.4 Slipstream Directives

To support slipstream mode the following directive is needed:

#pragma omp slipstream [type [[,] *tokens*]]]

The *tokens* specify the initial token count for synchronization, as shown in Figure 3.2. The initial token has a default value of zero. The *type* is either GLOBAL_SYNC, LOCAL_SYNC, or RUNTIME_SYNC. If not specified, the default value for the synchronization is implementation-dependent. In our implementation, we assume it to be global synchronization. Specifying RUNTIME_SYNC allows controlling the synchronization method at runtime using an environment variable similar to those used in the OpenMP standard. The environment variable name is proposed to be OMP_SLIPSTREAM. This environment variable takes the same arguments (*type* and

tokens) used in the SLIPSTREAM directive. The *type* argument may take an additional value of NONE, which disables running in slipstream mode.

This directive will affect the parallel region within which it is declared. Using this directive in the serial part is interpreted as a global setting for the program until being overridden by a later directive in the serial region. Using the directive on a parallel region takes precedence but does not override the global setting. Global settings are restored upon exiting the parallel regions.

This directive can be used in conjunction with conditional *if* statements, to restrict the use of slipstream only when the number of CMPs involved in solving the problem exceeds a certain limit.

4.2.5 Slipstream-aware OpenMP Compiler

In this work, we extended the Omni OpenMP [23] compiler to support slipstream execution mode. This compiler is freely available [1]. The compiler is supposed to work on IRIX 6.5 environment. We modified it to run on IRIX 5.3 available on our simulation environment.

4.2.5.1 Omni Compiler Overview

The Omni OpenMP compiler provides multiple thread models that can be used for implementing its internal library. Parallelizing a certain portion of a sequential code should have enough computation compared to the overheads of parallelization such as thread management, synchronization, and so forth. The Omni compiler tries to reduce the overhead of creating processes each time a parallel region is encountered. Instead process creation happens at the start of the program, and processes are kept in an idle pool. Parallel regions are transformed into functions by the compiler. The idle processes spins (on a flag), waiting for jobs by the master. When a parallel region is encountered, the master assigns the job indicated by the function representing the parallel region to a global variable, then sets the flags that indicate that a job is ready. A slave enters this parallel region by just calling the function indicated by the master. Based on the scheduling strategy, the slave may use its id to determine the portion

of work to execute or may serialize through a centralized entity to get information about its assigned job.

Omni compiler is an optimizing compiler for OpenMP. Among the objective of any optimizing compiler is to reduce parallelization overhead by adopting the following techniques:

1. Joining several regions into a single parallel region. There is an overhead associated with entering and exiting a parallel region. Under one model this may involve forking at entering and joining at exiting. If the implementation supports having a fixed slave pool technique, then entering a parallel region involves task distribution and exiting involves a synchronization.

2. Redundant barrier removal. Each work-sharing construct by default implies a barrier routine at the end of it, unless the *nowait* directive is specified. It is beneficial to remove barriers that are not needed for data dependency.

3. Privatize variables, if possible, to reduce the need for coherence overhead and to improve data locality. The OpenMP standard does not support privatization for dynamically allocated variables.

4. Determine the trade-off of parallel execution and parallelization overhead and sequential execution. Parallel execution may result in a slower execution if insufficient work exists. This also involves executing small loops using the *sections* construct. Although this optimization is highly desirable, it is difficult to achieve, as the iteration size of a loop is often determined by the problem size, which can be a runtime variable. Another dimension of the problem is that it may be worthwhile to parallelize a loop up to a certain degree of parallelism but the performance may degrade after that. The architecture can also be a decisive factor in determining communication overhead. Most compilers face a difficulty to reach a good decision at compile time.

To achieve these optimization goals, the Omni compiler uses an internal representation called parallel flow graph to model intra- and inter-process flows of data. This

information is used to reduce synchronizations and coherence overhead, and improve data locality.

4.2.5.2 Extending Omni Compiler

This study extends Omni 1.4a compiler for compiling programs parallelized with OpenMP directives. This compiler transforms C/F77 programs annotated with OpenMP directives into a multi-threaded C program with runtime library calls.

Figure 4.3 illustrates the stages of processing C/F77 files to produce an executable. We modified the Exc tools to support the new directive for slipstream mode to allow the programmer's hints to control slipstream behavior. Slipstream support also requires modifications to runtime libraries to support slipstream mode. Specifically, we modified the library for synchronization between threads forming A-stream R-stream pairs, constructs that control handling of parallel constructs, reduction variable handling, and task assignment. Other optimizations conducted by the compiler are not affected. So, the program transformation path to XObject is not affected except to map the slipstream directive to a library call. We choose the internal library based on UNIX shared memory model as it allows easier delineation between shared and local variables. Specifically, shared virtual address space is contiguous under this model. Other models would require more compiler involvement to guarantee no interleaving between shared and private spaces.

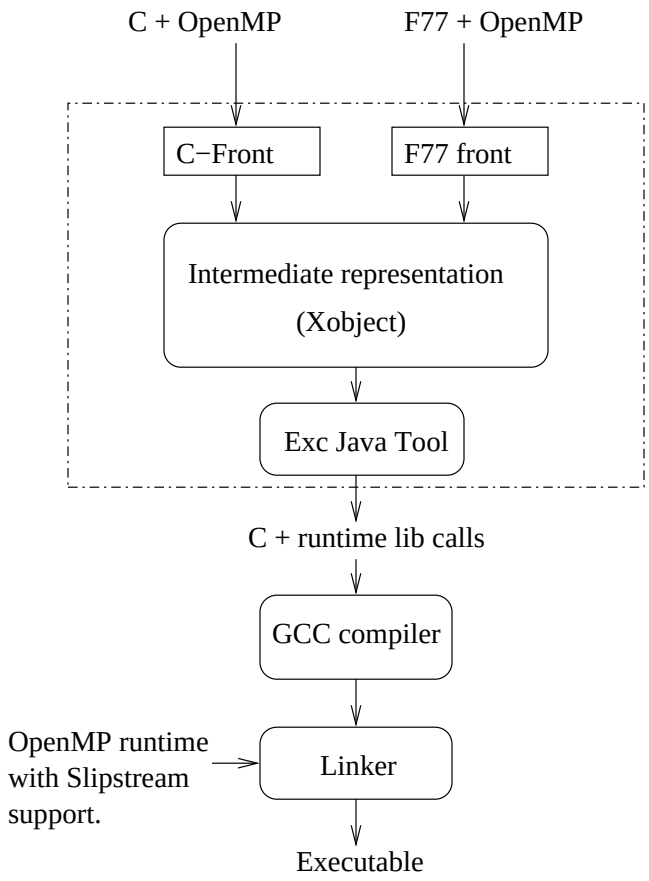


Figure 4.3: Overview of Omni OpenMP Compiler

Chapter 5

Slipstream-based Latency Hiding

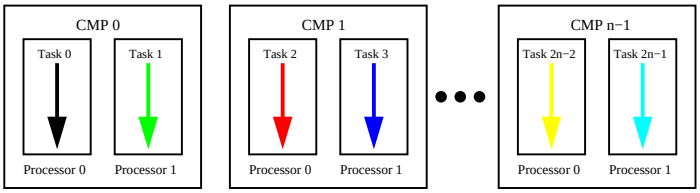
In this chapter, we discuss the use of slipstream for latency hiding and its performance. We investigate two techniques for slipstream-based latency hiding, prefetching and self-invalidation. We also evaluate the performance of prefetching in existence of dynamic scheduling for OpenMP benchmarks.

In our experiments, we compare three different modes of executions on a CMP chip. Figure 5.1 illustrates these three different modes of concurrent execution for a system with n CMPs: double, single, and slipstream.

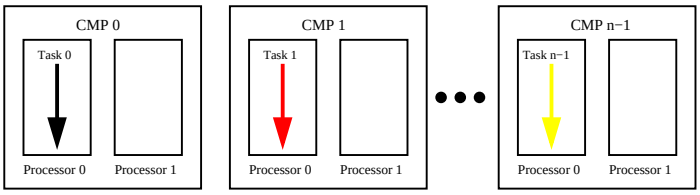
- In *double mode*, two parallel tasks are assigned to each CMP, one per processor, for a total of $2n$ tasks. This is the conventional execution model, maximizing the amount of concurrency applied to the program.

- In *single mode*, only one task is assigned to each CMP. One processor runs the task, while the other processor is idle. This can result in better performance than double mode when the scalability limit is approached. A single task means no contention for L2 cache and network resources on the CMP node. Also, fewer tasks means larger-grained tasks, which improves the computation-to-communication ratio.

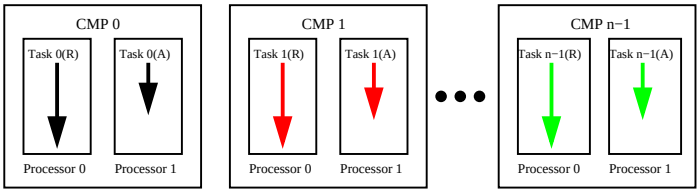
- In *slipstream mode*, two copies of the same task are created on each CMP, for a total of n task pairs. One processor runs the reduced task, or A-stream (short arrow), and the other runs the full task, or R-stream (long arrow).



(a) Double-mode execution.



(b) Single-mode execution.



(c) Slipstream-mode execution.

Figure 5.1: Execution modes for CMP-based multiprocessors.

5.1 Simulation Environment

To explore the performance of slipstream execution mode, we simulate a CMP-based multiprocessor. Each processing node consists of a dual-processor CMP and a portion of the globally-shared memory. Each CMP includes two processors. Each processor has its own L1 data and instruction caches. The two processors access a common unified L2 cache. System-wide coherence of the L2 caches is maintained by an invalidate-based fully-mapped directory protocol. The processor interconnect is modeled as a fixed-delay network. Contention is modeled at the network inputs and outputs, and at the memory controller. The system is simulated using SimOS [41, 16], with IRIX 5.3 and a MIPSY-based CMP model. Table 5.1 shows the simulated machine parameters, including the memory and network latency parameters, which

Table 5.1: System parameters of the simulated machine.

CPU	
MIPSY-based CMP Model	Clock Speed: 1.0 GHz
L1 Caches (I/D)	**L2 Cache (Unified)**
Size: 16 KB, 32B line Associativity: 2 Hit Latency: 1 cycle	Size: 1 MB, 64B line Associativity: 4 Hit Latency: 10 cycles
Memory Parameters(ns):	
BusTime: 30	Transit time from L2 cache to directory controller (DC)
PILocalDCTime: 10	Occupancy of the DC on a local miss
NIRemoteDCTime: 10	Occupancy of the DC on an outgoing remote miss
NILocalDCTime: 60	Occupancy of the local DC on an incoming remote miss
NetTime: 50	Fixed delay to transit the interconnection network
MemTime: 50	Fetch latency for the DC from local memory

are chosen to approximate the Origin 3000 memory system [45]. The minimum latency to bring data into the L2 cache on a remote miss is 290 ns, assuming no contention. A local miss requires 170 ns. The shared L2 cache manages coherence between its L1 caches and also merges their requests when appropriate.

Benchmarks used in this study are listed in Table 5.2. The first four benchmarks (SP, BT, MG, and CG) are an OpenMP port of NAS Parallel Benchmarks 2.3. The NAS NPB port to OpenMP, adopted in this study, is done by the Omni OpenMP compiler project [1]. Except for SOR, the others are taken from Splash-2 [52].

5.2 Slipstream Prefetching

A natural consequence of executing in slipstream mode is that the A-stream will prefetch shared data for the R-stream. Because the A-stream is executing the same task, it calculates the same addresses for shared data, and it loads that data before the R-stream. If the data is still valid when the R-stream reaches its load (i.e., not evicted or invalidated), then the R-stream will hit in the shared L2 cache.

Table 5.2: NAS-NPB 2.3 and SPLASH2 Benchmarks used in this study.

benchmark	**description**	**suit**
SP	3D Multi-partition for uncoupled systems of linear equations {scalar pentagonal(16×16×16)}.	
BT	3D Multi-partition for uncoupled systems of linear equations {block tridiagonal(24×24×24)}.	NAS NPB2.3
MG	Multigrid solver for Poisson equation(32×32×32).	
CG	Conjugate Gradient(1400).	
OCEAN	Eddy and boundary currents in influencing ocean movements (258×258).	
FFT	Fast Fourier Transform (64k complex doubles).	
LU	Parallel dense blocked LU factorization (512×512).	SPLASH2
WATER-NS	Molecular dynamic simulation, N-squared (512 mols).	
WATER-SP	Molecular dynamic simulation, Spatial (512 mols).	
SOR	Successive Over Relaxation (1024×1024).	

For coherence misses, prefetching is more likely to be effective when the A-stream is in the same session as the R-stream. In this case, loads from the A-stream will not violate dependencies imposed by synchronization. If the A-stream loads a line that is in the exclusive state, it retrieves the data from the owning cache and places it in the local L2 cache. Since this is a more expensive operation than a simple fetch from memory, the latency reduction seen by the R-stream is significant.

If the A-stream is in a different session, it has skipped one or more synchronizations, so its load may occur before the final store by the producer's R-stream. The premature load forces a loss of exclusive ownership by the producer's cache. This may degrade performance, because the producer must again acquire exclusive ownership to complete its stores. Furthermore, this invalidates the copy that was fetched by the A-stream, so the R-stream does not benefit. The same behavior can happen within the same session due to false sharing, where conflicting (unsynchronized) loads and stores may occur to different words in the same cache line.

As mentioned earlier, the A-stream task converts some skipped stores into exclu-

sive prefetches. The A-stream issues an exclusive prefetch if it is in the same session as the R-stream, and it is not in a critical section. The prefetch is likely to be effective, because the R-stream should be the only producer for that session. If the A-stream is not in the same session, or is in a critical section, then an exclusive prefetch is likely to conflict with R-stream accesses from the earlier session (or critical section). In this case, the store is simply skipped.

Because of the time-sensitive nature of prefetching, the choice of A-R synchronization model has a significant impact on its effectiveness. Global synchronization, with zero initial tokens, prevents the A-stream from entering the next session until all participating R-streams reach the barrier/event. Thus, A-stream loads will not occur until all producing R-streams for this session have finished writing. This reduces the number of premature prefetches, but it also reduces the opportunity to prefetch early enough to fully hide the latency from the R-stream. For applications with significant producer-consumer dependencies, this will likely be the best approach. Local synchronization allows the A-stream to move further ahead, subject to the number of allocated tokens. This more aggressive strategy will be useful for applications in which there is little actual sharing, and therefore few conflicting accesses.

5.2.1 Performance of Slipstream Mode with Prefetching only

The performance measure for the remainder of the paper will be speedup relative to single-mode execution (one task per CMP), because we are most interested in the region in which increasing concurrency is not an effective way to increase performance. Performance relative to single mode will easily show whether increasing concurrency (double mode) or increasing efficiency (slipstream mode) is more effective. But first we characterize the scalability of single-mode execution for our benchmarks.

Figure 5.3 shows the speedup for single mode over sequential execution for our nine benchmarks on 2, 4, 8, and 16 CMPs. There are three groups of benchmarks: those that continue to scale up to 16 tasks (Water-SP, LU, SOR), those that show signs of diminishing speedup (Water-NS, Ocean, MG, CG, SP), and one that shows decreasing performance (FFT). We expect slipstream mode to provide minimal benefit for the

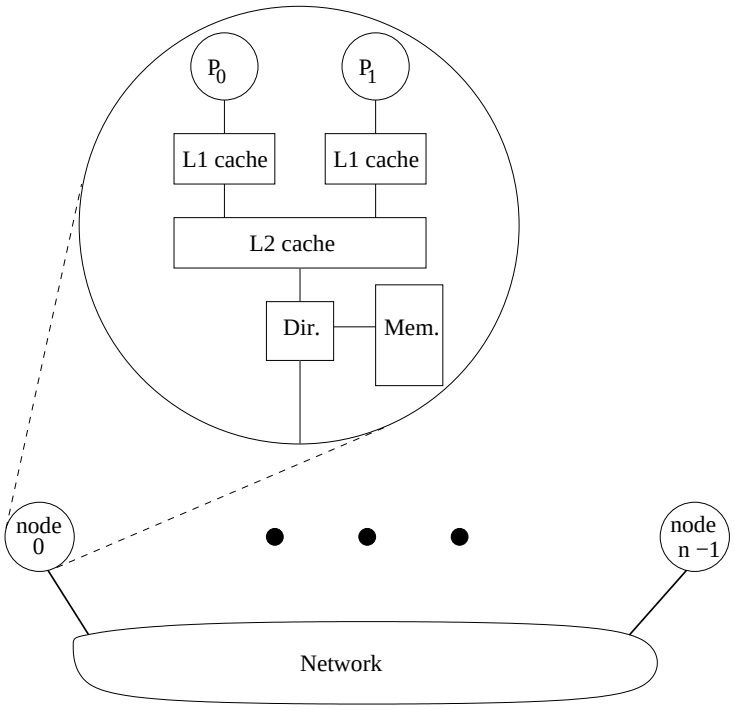


Figure 5.2: CMP based CC-NUMA system.

first group, since increasing concurrency will likely continue to improve performance at 16 CMPs. The second and third groups, however, may benefit more from slipstream mode than from doubling the number of tasks. Because of FFT's degrading single-mode performance, we will later only compare slipstream-mode performance at 4 CMPs or fewer.

Figure 5.4 shows the speedup of slipstream and double modes over single-mode execution. (To improve readability, double mode is shown only for 8 and 16 CMPs.) For slipstream mode, four different types of A-R synchronization are shown: (1) *one-token local (L1)*, which allows the A-stream to enter the next session when its R-stream enters the previous synchronization event; (2) *zero-token local (L0)*, which allows the A-stream to enter the next session when its R-stream enters the same synchronization event; (3) *zero-token global (G0)*, which allows the A-stream to enter the next session when its R-stream exits the same synchronization event; (4) *one-token global (G1)*, which allows the A-stream to enter the next session when its R-stream

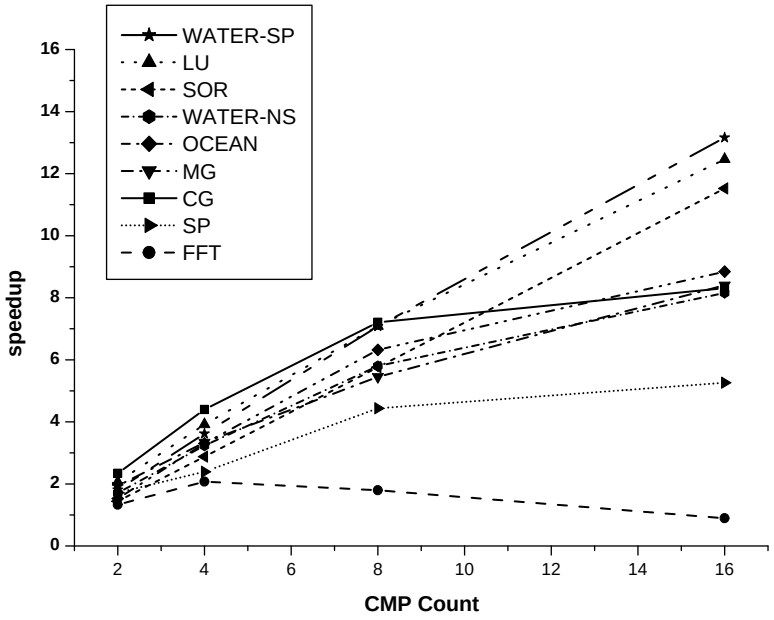


Figure 5.3: Speedup of single mode for 2, 4, 8, and 16 CMPs.

exits the previous synchronization event.

Consider the first group of benchmarks (LU, Water-SP, and SOR), which showed reasonable scalability. While slipstream shows some improvement over single for LU and Water-SP, it is much less effective than double for these configurations. In other words, there is still a significant amount of concurrency available at 16 CMPs, so slipstream mode is not the best choice. SOR, on the other hand, has apparently reached its scalability limit for this problem size, since double provides no benefit over single. Slipstream mode, however, performs 14% better than single mode.

For the remaining benchmarks, slipstream mode outperforms the best of single and double, beginning at four (FFT), eight (Ocean, SP), or 16 CMPs (CG, MG, SOR, Water-NS). At 16 CMPs, the performance improvement over the next best mode ranges from 12% (Ocean, MG) to 19% (Water-NS). For FFT, slipstream mode performs 14% better for 4 CMPs; further comparison is not shown because the absolute performance of FFT degrades at 8 CMPs and higher.

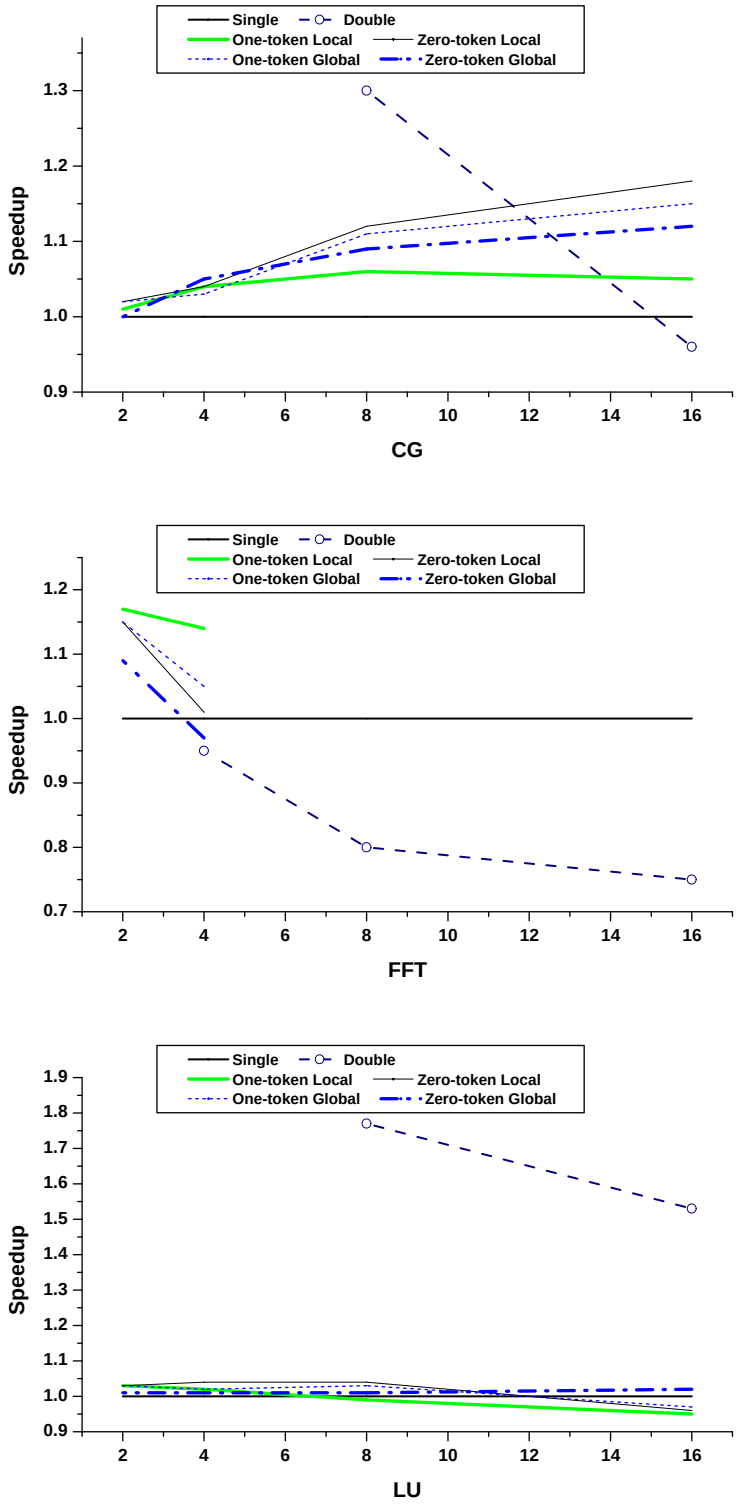
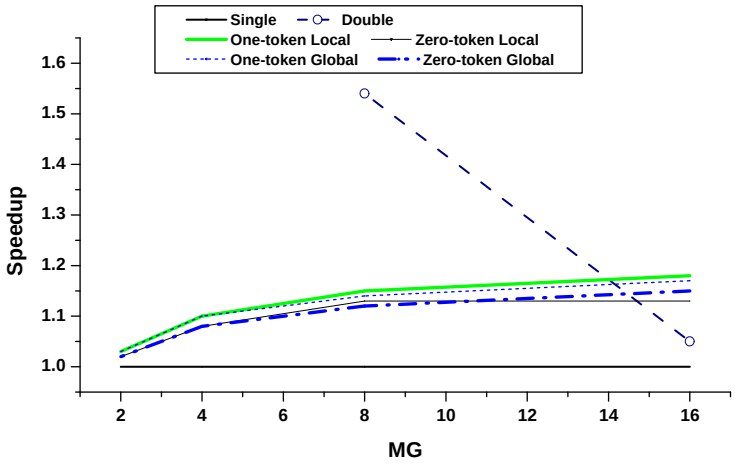
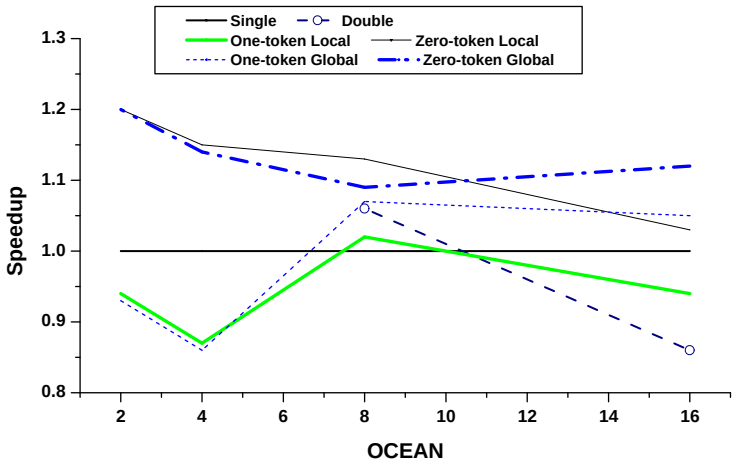


Figure 5.4: Speedup of slipstream and double modes, relative to single mode. For slipstream mode, four different types of A-R synchronization are shown.





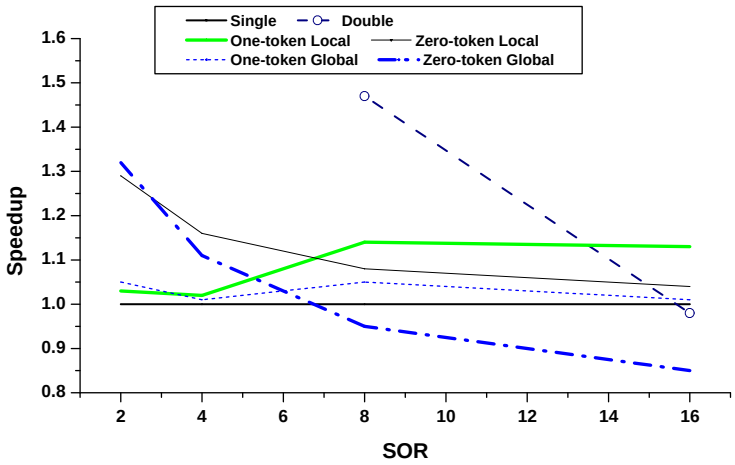
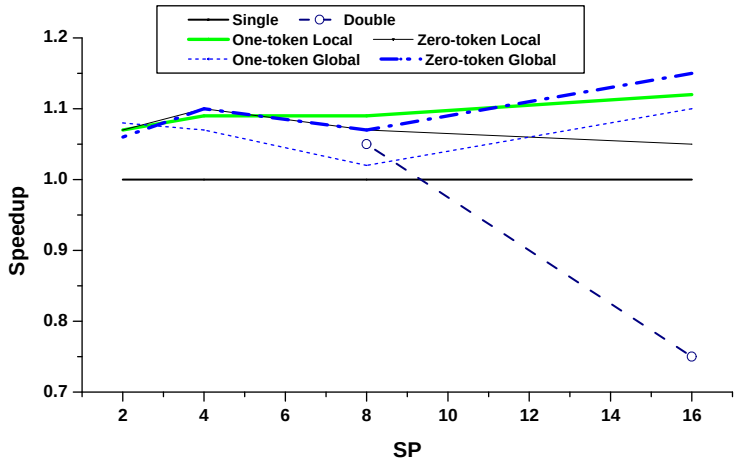
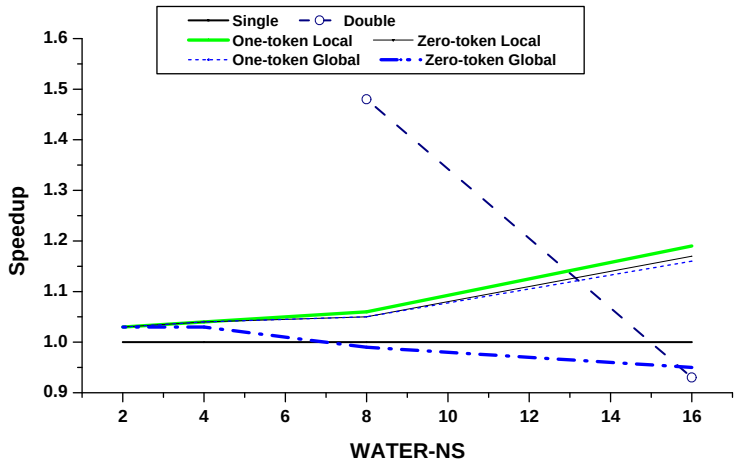


Figure 5.4: (*continued*) Speedup of slipstream and double modes, relative to single mode. For slipstream mode, four different types of A-R synchronization are shown.





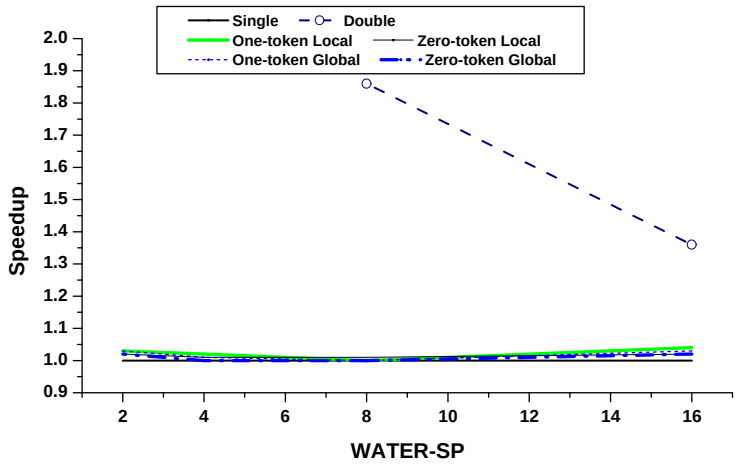


Figure 5.4: (*continued*) Speedup of slipstream and double modes, relative to single mode. For slipstream mode, four different types of A-R synchronization are shown.

There is no clear winner among the four A-R synchronization methods. In the seven benchmarks where slipstream mode delivers better performance, four benchmarks favor one-token local (FFT, Water-NS, MG, and SOR), two applications favor zero-token global (Ocean and SP), and one application favors zero-token local (CG).

Figure 5.5 shows the average execution time breakdown for single, double, and slipstream modes on a 16-CMP system. For slipstream mode, the time breakdown is shown for both the R-stream and the A-stream tasks, using the best-performing A-R synchronization method for each benchmark. Execution time is plotted relative to single mode. The time categories are busy cycles, memory stalls, and three kinds of synchronization waits – barrier, lock, and A-R synchronization. Reduction in stall time contributes to most of the gain achieved by slipstream mode. LU and Water-SP show little stall time (<8%) for single mode, which explains why slipstream does not help these applications. Interestingly, for SP and MG, slipstream mode decreases barrier time. The reason is that slipstream reduces the imbalance due to variability of memory access latency between barriers. A-R synchronization time is an indication of how much the A-stream is shortened, relative to the R-stream. If the A-stream is far ahead, then it will often wait for the R-stream to end its current session and give permission for the A-stream to proceed.

Figure 5.6 shows the breakdown of memory requests for shared data for slipstream mode with different synchronization methods. Shared memory requests generated by the A-stream are divided into three categories. An *A-Timely* request brings data into the L2 cache that is later referenced by the R-stream. For *A-Late*, the same data is referenced by the R-stream before the A-stream request is satisfied. If data fetched by the A-stream is evicted or invalidated without being referenced by the R-stream, the reference is labeled as *A-Only*. The third component is considered harmful, as it reflects an unnecessary increase in network traffic and may slow down applications due to unneeded data migration. Memory requests by the R-stream are divided into similar categories: *R-Timely*, *R-Late* and *R-Only*. The top graph in Figure 5.6 shows the breakdown for read requests, and the lower graph shows the breakdown for exclusive requests.

Exclusive requests by the A-stream are due to converting some shared stores into

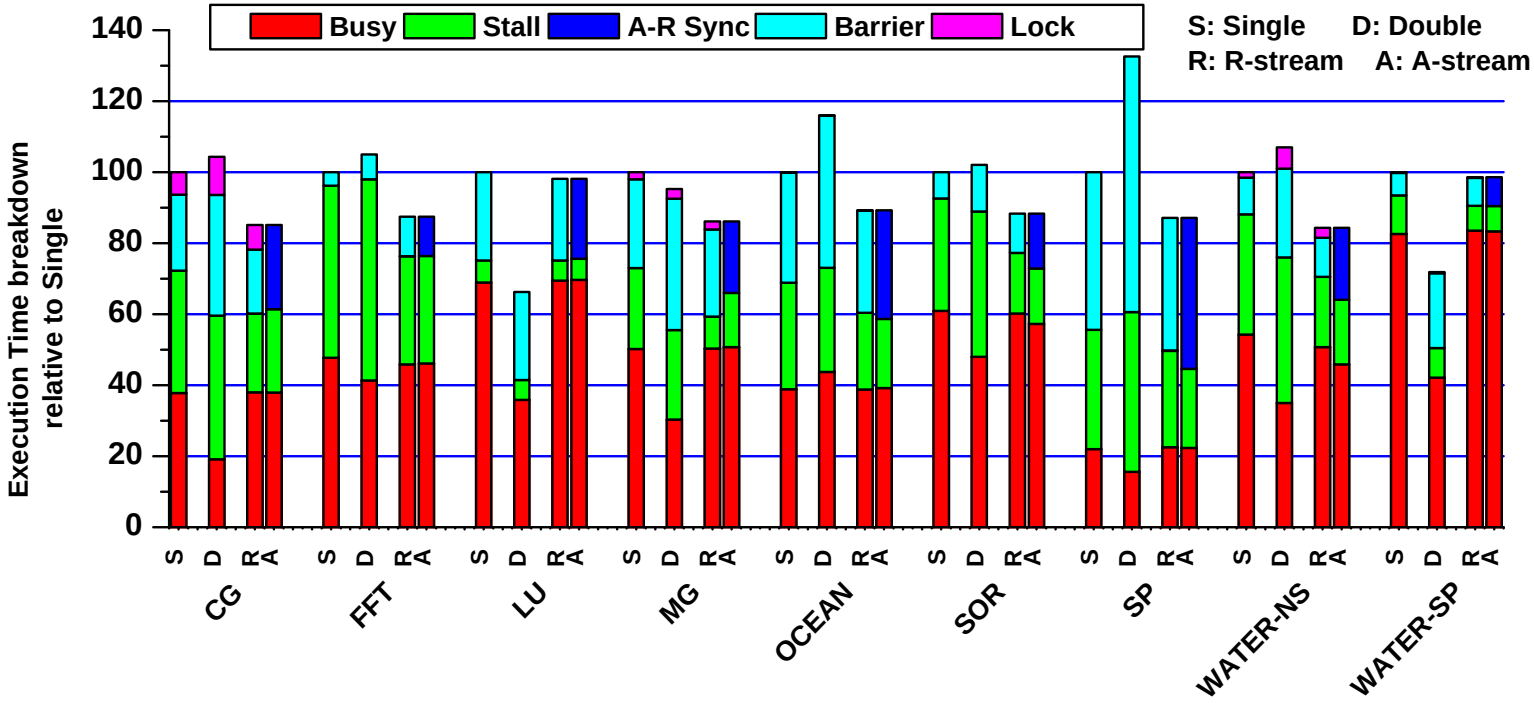


Figure 5.5: Execution time breakdown for single (S), double (D), and slipstream modes (A and R). Execution time is relative to single mode. The best A-R synchronization method is used for slipstream mode.

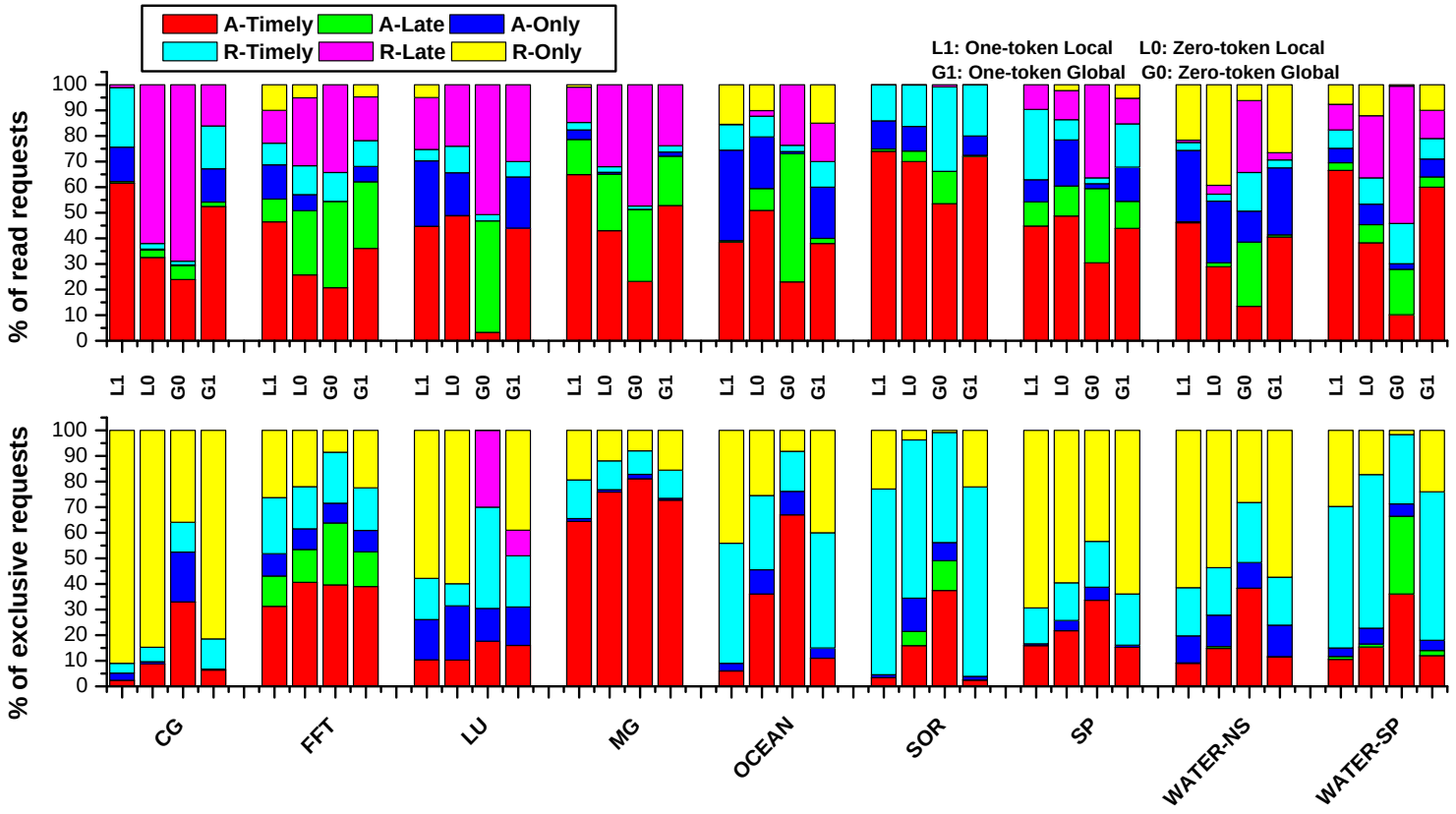


Figure 5.6: Breakdown of memory requests for shared data.

prefetches. This conversion only happens when the A-stream is in the same session as the R-stream and is not in a critical section. This explains the low percentage of useful exclusive prefetches for most applications (27% on average).

The request breakdowns highlight the differences between tight and loose A-R synchronization. Zero-token global (G0) is the tightest synchronization model, and one-token local (L1) is the loosest.

Zero-token global exhibits the lowest fraction of A-Timely read requests (22% on average), because the A-stream is not allowed to run very far ahead of the R-stream. This is also reflected in the high rate of A-Late requests (27% for reads, 7% for exclusive). On the other hand, it has the largest fraction of A-Timely exclusive requests (43%). The reason is that stores are converted to exclusive prefetches when the A-stream is in the same session as the R-stream; this is more often the case with tight A-R synchronization.

One-token local, the loosest synchronization, has the highest rate of A-Timely read requests (54% on average), a low fraction of A-Late requests (4% reads, 1% exclusive), and the lowest rate of A-Timely exclusive requests (17%). Because the A-stream is allowed to run very far ahead of the R-stream, its read requests are more likely to be satisfied before the R-stream needs the data. But it is less likely to be in the same session as the R-stream, so the opportunity for exclusive prefetching is lower. This also results in the highest fraction of premature (A-Only) read requests (16% on average).

Each synchronization method has its good and bad attributes, and the resulting performance is application-dependent. For example, Ocean benefits more from tight synchronization (zero-token global), as it has negligible premature read requests (A-Only) compared with other synchronizations. The A-stream also provides a higher rate of successful (A-Timely) exclusive requests. FFT favors loose synchronization (one-token local), as it provides more timely read requests than other methods, and nearly as many timely exclusive requests.

The R-stream also sends requests to memory. Data loaded by the A-stream may be invalidated or evicted, and some exclusive data requests will be skipped entirely by the A-stream. While R-Timely, R-Late, and R-Only components do not directly

reflect performance, they complete the view of how much correlation exists between the shared data referenced by both streams. The highest correlation (reflected by small R-only and A-only components) is associated with tightest synchronization, zero-token global. For this synchronization, 98% of read requests and 77% of the exclusive requests are for data that is referenced by both streams.

We have only considered a single, static choice of A-R synchronization for each execution, but there are several possibilities for more flexible approaches. For example, the first few iterations of a program may be used to explore each synchronization option and settle on the best performer for the remainder of the program. Alternatively, a dynamic monitor can collect session statistics and adaptively adjust the synchronization method. Finally, the appropriate methods for specific kernels within an application may be known by the programmer and can be annotated in the library or source code.

To summarize this section, slipstream-based prefetching can be supported with minimal hardware changes on CMP-based multiprocessors. For seven of the nine benchmarks, prefetching alone improves performance by 12-19% over the next best mode (single or double) on a 16-CMP system. However, prefetching is only the simplest optimization enabled by slipstream mode. We can use the sharing predictions provided by slipstream mode to better optimize coherence traffic, as described in Section 5.3.

5.2.2 Slipstream Prefetching with Dynamic Scheduling

Interaction between slipstream mode and dynamic/guided scheduling has several interesting aspects. First, being ahead for A-stream relies mostly on skipping shared memory operations and not on skipping synchronization, as discussed in Section 4.2.3.2. Second, the synchronization between A-stream and R-stream will be tighter than global-zero, as there is an additional synchronization at the scheduling points. Finally, these scheduling techniques can increase cache miss rate, compared with static scheduling, due to the possible data migration.

The behavior of dynamic/guided scheduling relies on scheduling parameters, such

as chunk size. The choice of this parameter is dependent on iteration count, degree of parallelism, and the underlying hardware. In the studied benchmarks (except for CG), parallelized loops are of coarse-grained and few in number. So, we used the compiler defaults for all application, except for CG, where we used chunk size equal to half the the assignment under static block assignment. While this does not necessarily give the best performance, it captures two main properties we want to investigate. The first property is the existence of multiple scheduling decisions and thus multiple synchronization points between barriers. The second property is the possible data migration due to these scheduling policies.

We used the same OpenMP benchmarks used in the previous section. In our experiment, we noticed that the performance for most of the benchmarks degrades with dynamic scheduling, as the scheduling overhead is high. Additionally, these applications are iterative and data are reused across iterations, so they lose the advantage of using cached data if data migration occurs. Finally, most of these benchmarks have a big granularity of parallel work, which is not the best candidate for dynamic scheduling.

We conducted the comparison with one task/CMP only, as the overhead of scheduling increases substantially with the increase of the number of processes as well as the data migration. We used only zero-token global synchronization mode for slipstream. This is because there are additional synchronizations at scheduling points that make other slipstream synchronizations converge to zero-token global.

Figure 5.7 shows the execution time break down for our benchmarks based on dynamic scheduling. The scheduling overhead for the base case has an average of 11%. The stall time to the busy time ratio also increased compared with static scheduling. Figure 5.8 shows the breakdown of memory requests for shared data for slipstream with dynamic scheduling. For shared data read, the A-timely component is 28% on average and the A-late component is 26% on average. For read exclusive requests, the A-stream provides good coverage (59% in average for A-timely, and 2% for A-late). Slipstream mode improves the performance of the base mode due to the high contribution of the stall time to the total execution time. The improvement ranged from 5% for MG to 20% for SP.

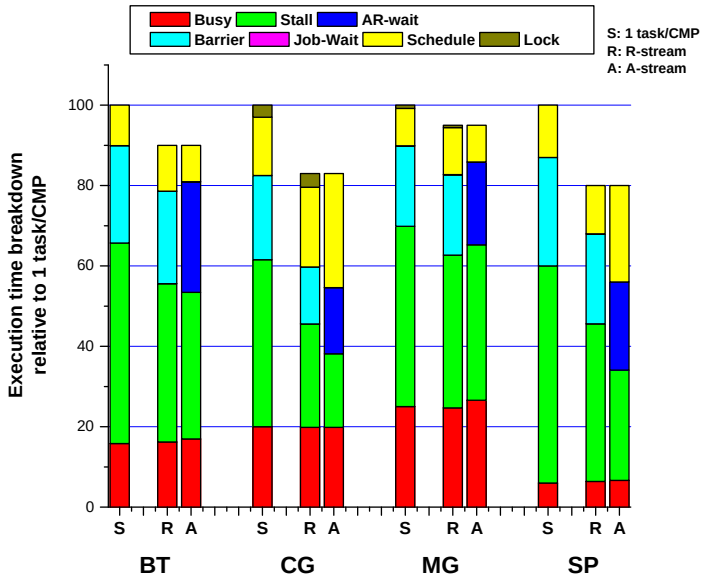


Figure 5.7: Execution time breakdown for single, zero-token global slipstream mode (dynamic scheduling).

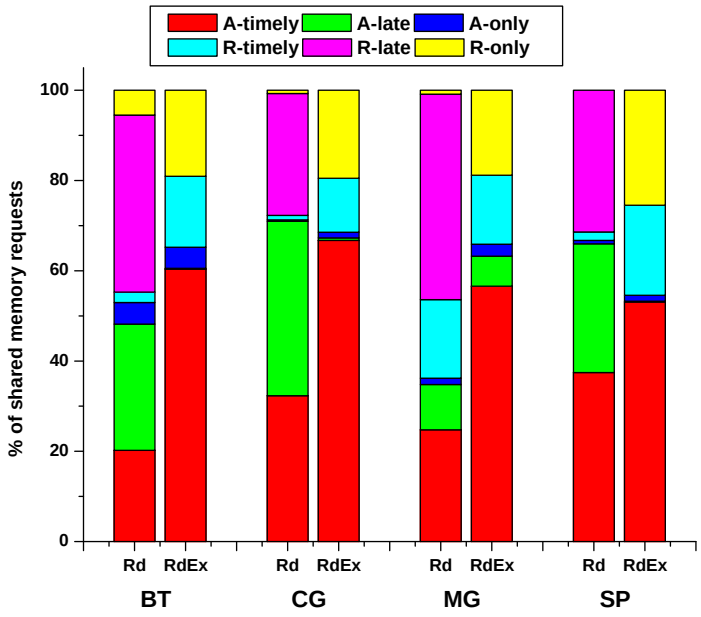


Figure 5.8: Breakdown of memory requests for shared data (dynamic scheduling). Slipstream synchronization is zero-token global.

While dynamic scheduling does not prove to be a good choice for these benchmarks, we show that the potential for slipstream mode to improve performance is high with dynamic scheduling as the memory stall time component is usually high.

5.3 Slipstream Self-invalidation

Coherence traffic is difficult to optimize using prefetching alone, because there is a timing component that is not easily captured by local access information. If a line is in the exclusive state, prefetching too early will cause useless traffic and latency if the producer has not yet performed all of its stores. Also, data protected by critical sections are difficult to prefetch effectively. When it is difficult to guarantee successful prefetches, we can utilize the accurate information provided by the A-stream about future R-stream accesses to more efficiently manage operations on shared data. There are a number of such optimizations that can be implemented using slipstream. In this paper, we investigate *self-invalidation* (SI) [27, 25] as a technique to reduce the latency of coherence misses.

Self-invalidation advises a processor to invalidate its local copy of a cache line before a conflicting access occurs. When successful, this will reduce invalidation messages and writeback requests. A subsequent load from another processor will find the data in memory, rather than having to request a writeback from the owning cache. A subsequent store will acquire an exclusive copy from memory without having to invalidate copies in other caches.

We investigate a form of SI to enhance slipstream's ability to prefetch. To reduce premature prefetches, we introduce a new type of memory operation called a *transparent load*, issued by the A-stream. A transparent load may return a non-coherent copy of the data from memory without adding the requester to the sharing list. Since the request is from an A-stream, the memory controller adds the requesting node to a *future sharer* list. The future sharing information is used to send a self-invalidation hint to the exclusive owner of the cache line. This causes the owning cache to write the data back to memory when its last write is complete, moving the data closer to the consumers or a new producer.

Sections 5.3.1 and 5.3.2 describe the slipstream variants that use transparent loads and SI, and Section 5.3.3 describes the resulting performance.

5.3.1 Transparent Load

A transparent load is designed to prevent the premature migration of shared data due to an A-stream prefetch. The A-stream uses transparent loads to satisfy a read miss, if it is one or more sessions ahead of the corresponding R-stream or when it is inside a critical section. Under these conditions, it is more likely that an A-stream may load data before its final value has been written.

If a transparent load finds the line in the exclusive state at the memory, the memory sends a transparent reply, containing its current (possibly stale) copy of the data without requiring a writeback from the owning cache. The load is transparent to other processors in the system, because the requester is not added to the coherence protocol's sharing list. This means that the requester's copy of the cache line will not be invalidated due to a store by another processor. Therefore, when the transparent reply arrives, the line is marked as "transparent" in the L2 cache. The data is then visible only to the A-stream, not to the R-stream. This prevents the R-stream from reading non-coherent data, yet allows the A-stream to continue making forward progress.

If the line is found in a non-exclusive state (shared or idle), the transparent load is upgraded to a normal load, and the requesting node is added to the sharing list. A normal reply is sent, and the cached data is available to both the A-stream and the R-stream.

As mentioned earlier, the A-stream issues normal loads only if it is in the same session with its R-stream and not within critical sections. In this case, it is presumed that the prefetch is not premature, because the synchronization dependency has been respected. Thus, it is more beneficial to perform a normal load, to retrieve the data from the owning cache and bring it into the local cache for the R-stream's benefit.

5.3.2 Future Sharers and Self-invalidation

Transparent loads decrease the penalty due to premature prefetches of shared data, but they also remove one of the benefits of prefetching: forcing the producer to write back its cache line in anticipation of a subsequent load. We want to enable a timely writeback, one that moves the data closer to the requesting R-stream but that does not require the producer to lose ownership until it has finished with the line. For this purpose, we use A-stream transparent loads as hints of future sharing behavior, and we use these hints to implement a mechanism for SI. Our approach is illustrated in Figure 5.9.

In the top half of the figure, the memory directory receives a transparent load request for a line in the exclusive state (1). The directory sends a transparent reply to the requester and a self-invalidation hint to the cache that owns the exclusive copy (2). It adds the requester to its future sharer list.

In the bottom of Figure 5.9, the directory receives a transparent load request for a line in the shared or idle state (1). An upgraded (normal) copy of the cache line is sent (2), and the requester is recorded both as a sharer and as a future sharer. Later, when an R-stream sends a read-exclusive (or upgrade) request for the line (3), the directory invalidates shared copies (4,5) and includes a self-invalidation hint with the reply to the requesting R-stream (6).

The future sharer bit for a node is reset whenever the cache line is evicted from that node, or when any request from the R-stream reaches the directory. This allows the future sharing information to be persistent enough to be useful for migratory data, which is written by multiple nodes, yet not so persistent that it fosters many unnecessary self-invalidations.

Self-invalidation hints are recorded by the owning cache. Following the heuristic of Lebeck and Wood [27], lines marked for self-invalidation are processed when the R-stream reaches a synchronization point. Unlike their approach, marked lines are either self-invalidated or just written back, based on the code in which they were accessed. If a write access occurs within a critical section, the line is invalidated (assumed migratory). Otherwise, the line is just updated in memory, and ownership

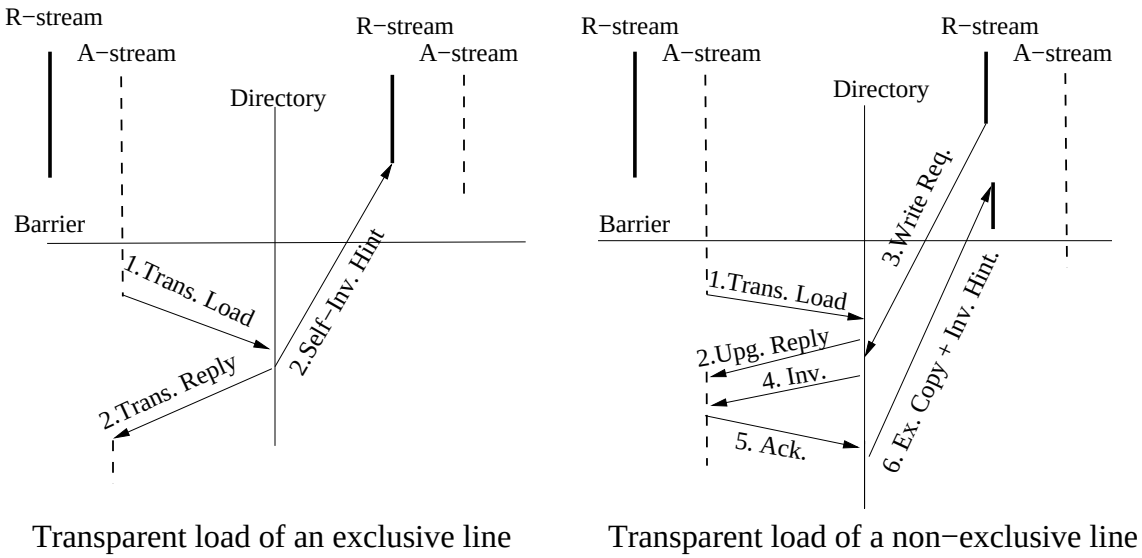


Figure 5.9: Slipstream-based self-invalidation.

is downgraded from exclusive to shared (assumed producer-consumer). Invalidations are performed asynchronously, overlapped with barrier or unlock synchronization, and initiated at a peak rate of one every four cycles. Lai and Falsafi [25] advocate a more timely self-invalidation, following the producer's predicted last touch of the cache line. This approach can be implemented in slipstream if explicit access predictions are passed from the A-stream to its R-stream. We will address this capability in Chapter 6.

5.3.3 Performance of Transparent loads and Self-invalidation

To evaluate the performance of transparent loads and Self-invalidation (SI), we focus on the 16-CMPs configuration for all applications except for FFT where we use a 4-CMPs configuration. To achieve a balance between accuracy and having a view of the distant future, we use one-token global A-R synchronization. We exclude LU and Water-SP, as these benchmarks do not have the potential of improving from slipstream mode due to their small stall time and the large benefit they gain from more concurrency.

Figure 5.10 shows the percentage of A-stream read requests issued as transparent

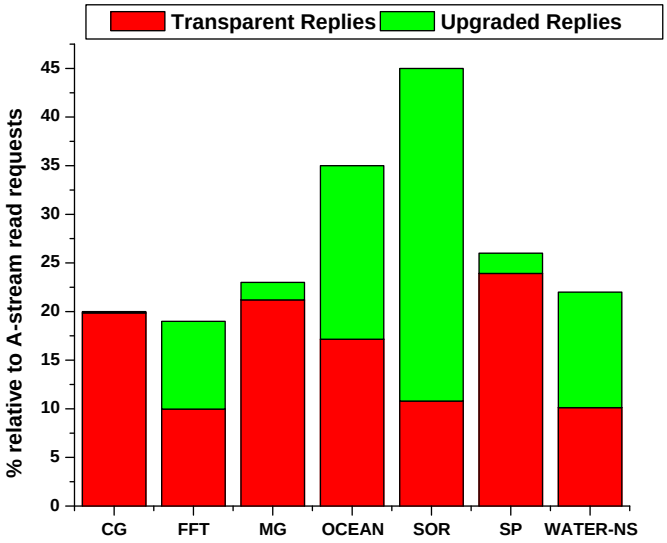


Figure 5.10: Transparent load breakdown.

loads and the breakdown of these transparent loads into those that receive a transparent reply and those that are upgraded. For the benchmarks tested, 19% to 45% (average 27%) of read requests initiated by the A-stream are issued as transparent loads. On average, 59% of transparent loads receive transparent replies, and the remaining 41% are upgraded into normal loads.

In Figure 5.11, we categorize all performed self-invalidations as either correctly predicted or mispredicted. (We will use self-invalidations here to refer to both invalidations and downgrades of the cache line by the owning cache.) A mispredicted self-invalidation is one in which the owning R-stream references the same line again (requiring the same local coherence state) in the next session. Prediction accuracy ranges from 86% (SP) to 96% (CG). This high accuracy is due to the high correlation between A-stream and R-stream memory references. Most of the mispredictions are caused by false sharing, where multiple caches access the same line and at least one requests an exclusive copy without explicit synchronization. We also show the number of self-invalidations that should have been performed, but were not. This is indicated when an R-stream L2 miss reaches the directory and finds the line in the exclusive state. The not-predicted percentage is insignificant for most applications.

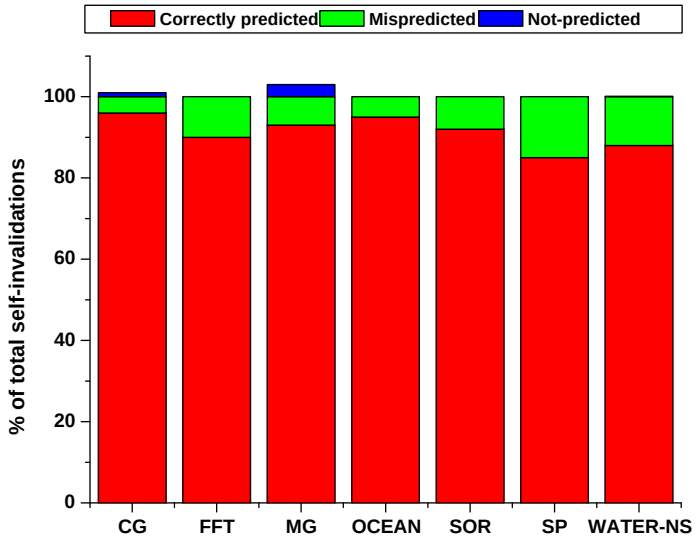


Figure 5.11: Accuracy of slipstream-based SI.

Figure 5.12 shows the speedup of slipstream mode over the best of single and double for three slipstream configurations. The first slipstream configuration does only prefetching, as described in the previous section, using one-token global synchronization. Next, transparent loads are added, without SI. In some cases (FFT, MG, and SOR), using transparent loads decreases performance because of the reduction in prefetching. For CG, Ocean, SP, and Water-NS, however, the elimination of premature prefetches results in a 4% increase in speedup. When transparent loads and SI are combined, there is an additional speedup of 6% for Ocean, 8% for SOR, 9% for FFT, 12% for Water-NS, 14% for CG, and 15% for SP. There is 4% less speedup for MG compared to slipstream with prefetching only. MG has a low percentage (about 4%) of premature read requests (A-Only in Figure 5.6), which indicates there are few dependency violations, while about 21% of A-stream requests are handled transparently when self-invalidation is enabled (Figure 5.10). Self-invalidation yields less benefit when prefetching works well, because prefetching hides more latency by bringing the line into the consumer's cache.

The above comparisons considered only one-token global synchronization, rather than using the method that results in the best prefetching-only performance. Com-

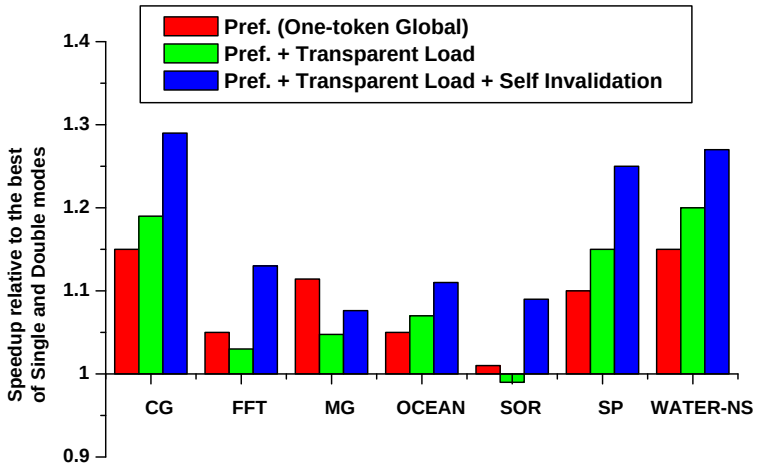


Figure 5.12: Performance of slipstream with transparent loads and SI, relative to the better of single and double modes; A-R synchronization is one-token global.

pared to the best prefetching configuration for each benchmark (from Section 5.2), SI provides additional speedup for Water-NS (8%), SP (10%), and CG (11%). For SOR, FFT and Ocean, SI does not provide a significant improvement over the best slipstream prefetching-only method.

5.4 Possible Enhancements

The above discussion provides insights about how to improve the proposed scheme. We notice that there is a tension between using conservative synchronization (thus getting less premature, but not timely data) and aggressive synchronization (thus getting ahead enough, but with possible premature data). This complicates deciding the optimal synchronization method between A-stream and R-stream.

This tension can be relaxed if we allow more aggressive instruction removal while keeping the synchronization conservative. We removed only shared stores from A-stream references. These dead stores usually lead to many dead floating point operations. The removal of these expensive operations can allow the A-stream to be ahead enough, even under conservative synchronization. This may also require providing more synchronization within a session.

Chapter 6

Identification of Last Access or Write

In this chapter, we discuss the accuracy and the implementation cost of different schemes for specifying last touch or write of a cache line. This identification helps in triggering timed and preemptive coherence actions, such as self-invalidation.

We introduce different last touch identification patterns. We call these identifying patterns signatures. These signatures can be divided into groups. The first group is based on instruction streams and program behavior. The second group is based on the temporal property of access streams. We also discuss the suitability of these schemes to different kinds of optimizations. We finally introduce a mechanism to collect PC-based signatures.

6.1 Importance of Identifying Last Access or Write

Optimizations like self-invalidation, barrier speculation, and power management are based on detecting when a cache line is ready for special actions. These actions can be invalidation, allowing migration, or turning off a cache line to save energy. Specifying that a line is ready for these special actions is triggered by a pattern. We call this special pattern a last access signature, or simply a signature. A signature is a pattern of accesses/events that triggers an action.

Depending on the optimization type, we may need signatures that indicate last write or last touch. For instance, speculation after barrier synchronization benefits from specifying the last write of a cache line to allow data migration to speculative threads. This migration of data happens when a speculative thread tries to write a cache line. For self-invalidation, last write access can be used to determine when to update lines to the memory for future readers. Last touch (read or write) can be used to support speculation after barrier synchronization if expiring copy is not supported and also can be used with self-invalidation for migratory data.

In this work, we focus in developing and assessing the cost and performance per session (*i.e.* between two barrier calls). This kind of analysis can simplify the signatures collected but on the other hand make the results presented not suitable for power optimization. The reason is that cache lines may be accessed in consecutive sessions. Obviously, the decision of a last access in a certain session does not guarantee that the line will not be accessed for a sufficiently long time (*i.e.* in future sessions).

The objective is to achieve a good signature that gives both accurate results and low cost.

6.2 Assessing the Quality of a Signature

The accesses to a certain address can be grouped as a sequence of patterns, as shown in Figure 6.1. Each pattern resembles a proposed last access signature pattern. A good signature scheme will ensure that a certain signature pattern will only occur once at the end of the access patterns.

Two types of ambiguity, as shown in Figure 6.1, can indicate the quality of a certain signature. The first type of ambiguity is defined as the presence of multiple instances of the last access pattern in the accesses for a single cache line. We call this type of ambiguity *local ambiguity*. This ambiguity indicates that this signature pattern may lead to incorrect prediction of the last access. The second type of ambiguity occurs if the last access pattern of one address appears as a non-last access pattern for another address. We call this kind of ambiguity *global ambiguity*. This kind of ambiguity

prevents using last touch signature patterns independent of addresses.

For a good last touch signature, local ambiguity should not exist. A signature that does not suffer from global ambiguity has a lower implementation cost, as identifications of these last touch patterns can be independent of addresses. Thus the buffering requirement for these patterns is independent of the data set size.

The complexity of identifying last touch is due to the following factors:

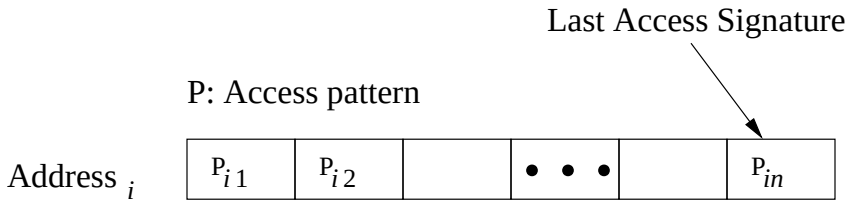
- Cache line size that contains multiple words. If every word is accessed only once then the line will receive multiple accesses.
- Loops that are used to optimize code size for accessing large arrays. These loops make the same program counter responsible for accessing many addresses.
- Conditional branches, which make some of the accesses conditionally executed.
- Higher level optimizations like tiling (or blocking) that try to optimize for good cache locality. These optimizations increase the nesting level for loops.
- Having multiple phases of accesses for the same data in the same session. This is an algorithm-dependent behavior.

6.3 PC-Based Last-Touch Prediction Schemes

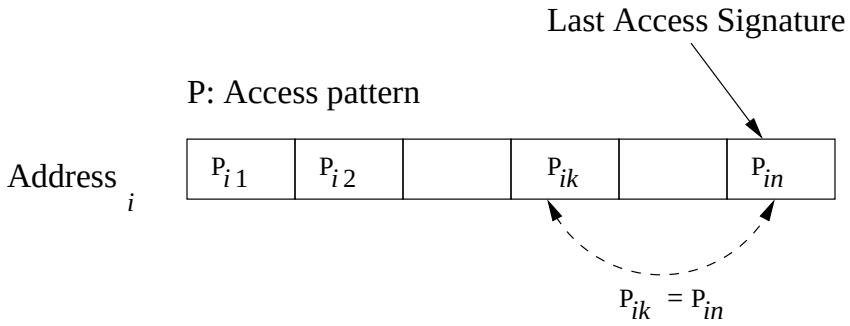
An earlier proposal by Lai and Falsafi [25] investigated the use of the PC or a sequence of PCs to predict the last touch of a cache line. This last touch prediction is used to trigger self-invalidation.

We extend the search for a last touch signature, based on the above discussion, using the following components:

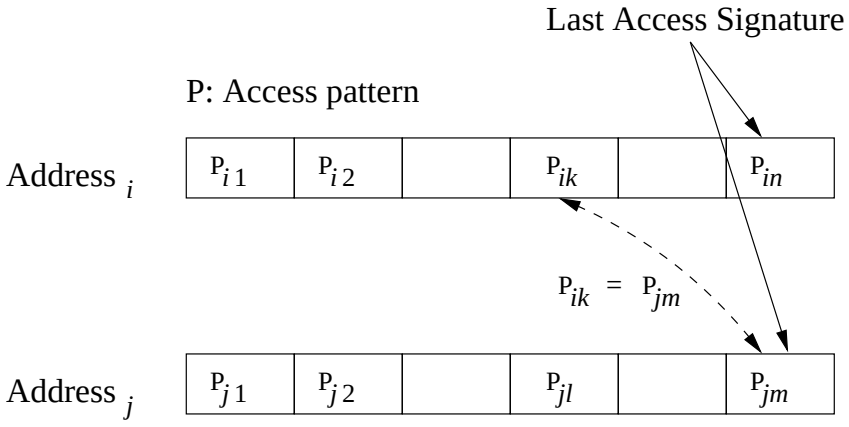
- Access program counter (PC)
- Access sequence of PCs
- Access word number within a cache line



(a) Last access signature



(b) Local ambiguity



(c) Global ambiguity

Figure 6.1: Last access/write definition and different ambiguity types.

C source	MIPS assembly
for(i=0; i<10; i++) a[i] *= 2;	**PC Label Instr** 0x100 lw v1,0(gp) # addr of a 0x104 addu a0,zero,zero 0x108 L1: lw v0,0(v1) 0x10c addiu a0,a0,1 0x110 sll v0,v0,1 0x114 sw v0,0(v1) 0x118 slti v0,a0,10 0x11c bne v0,zero,L1

Array a is cache line aligned, word = 4 bytes, cache line size = 32 bytes

	cache line 0								cache line 1	
PC	0x114	0x114	0x114	0x114	0x114	0x114	0x114	0x114	0x114	0x114
Access	0	4	8	12	16	20	24	28	0	4
Br	N	N	N	N	N	N	N	N	N	Y

Signature Scheme	Signatures	Local ambiguity	Global ambiguity
PC	0x114	Yes	Yes
5 PCs	(0x114,0x114,0x114,0x114,0x114), (0x114,0x114,*)	Yes	Yes
(PC, Access)	(0x114, 28), (0x114, 4)	No	Yes
(PC, Branch)	(0x114, N), (0x114, Y)	Yes	Yes
(PC, Access, branch)	(0x114, 28, N), (0x114, 4,Y)	No	No

Figure 6.2: Ambiguity associated with different signature schemes (considering only write accesses) for a simple loop.

- Conditional branch outcomes following the access
- Number of accesses to a cache line

Based on the above disambiguation components, we investigate the following last touch/write signature patterns:

1. Single PC
2. PC and access word number
3. PC and 8 conditional branch outcomes

Table 6.1: Benchmarks used and their problem sizes.

benchmark (SPLASH2)	**size**	**benchmark (NAS-OMP)**	**size**	**benchmark (SPEC-OMP2001)**	**size**
LU(C&NC)	512×512	MG	32×32×32	swim_m	384×384
OCEAN(C&NC)	258×258	SP	16×16×16	wupwise_m	8×8×8×8
FFT	64k	BT	16×16×16	equake_m	7294
WATER-NS	512 mols	CG	1400	mgrid_m	32×32×32
WATER-SP	512 mols			applu_m	16×16×16

4. PC, access word number, and 8 conditional branch outcomes

5. A sequence of 5 PCs

Figure 6.2 shows a simple loop iteration and the corresponding last write signatures due to different schemes. In this example, we show that a signature scheme of (PC, access word, branch outcome) can provide no local or global ambiguities where other schemes fail to achieve the same objective.

Table 6.1 shows the applications investigated in this chapter and their problem sizes. For all signatures, we measured the percentage of local and global ambiguities. The percentages are computed based on the number of signatures that suffer these ambiguities compared to the total signature count.

Figure 6.3 shows both local and global ambiguities, considering all accesses. Figure 6.4 shows local and global ambiguity percentages considering write accesses only. Table 6.2 summaries the average percentage of ambiguity based on the schemes proposed. As expected, the single PC signature scheme produces high level of ambiguity both locally and globally because of loop constructs. The average percentage for local ambiguity is 68% considering writes and 71% considering all accesses. Including the access word number is the second best choice of signature disambiguation. A sequence of PCs does not provide a good disambiguation mechanism due to cache line granularity (64 byte cache block, in this study). The best disambiguation signature is a combination of PC, access word number and conditional branch outcome. The average percentage of local ambiguity is 8% and global ambiguity is 6% for writes (12%

Table 6.2: Average percentages of signature ambiguities for all applications. Percentages within bracket exclude LU (C&NC) applications.

Signature type	**Write accesses**		**All accesses**	
	Local Ambiguity	**Global Ambiguity**	**Local Ambiguity**	**Global Ambiguity**
PC	67.9 (65.4)	20.5 (22.4)	71.3 (71.1)	32.6 (35.3)
PC + Access	12.9 (5.7)	33.7 (37.5)	27.3 (22)	36.7 (40.4)
PC + Branch	27.5 (23.1)	38.5 (40.1)	28.3 (24.7)	39.2 (41.5)
PC + Access + Branch	7.7 (1.9)	5.5 (5.6)	12 (6.4)	9.6 (10.3)
5 PCs	74.3 (74.2)	17.5 (19.9)	75.9 (72.8)	30.3 (31.8)

and 10%, respectively for all accesses). Removing LU from the averaging reduces local ambiguity to 2% for writes (6% for all accesses). The two algorithms investigated for LU have high ambiguity due to the multiple phases of producing the same word within a session. These phases of execution involve calling the same function many times. For SOR, OCEAN, MG, SP, BT, CG, wupwise_m, swim_m, mgrid_m, and applu_m applications there is 0% local ambiguity for writes. These applications also have global ambiguity of 1.5%. It is also noticeable that the last write can be disambiguated more easily than the last access (read or write).

It is noticeable that adding access word number to the PC reduces the local ambiguity but does not reduce the global ambiguity. Additionally, adding branch outcomes to reduce the global ambiguity. Adding branch outcomes alone to the PC suffers both local and global ambiguity. This conforms with the intuition given by the example in Figure 6.2.

6.3.1 Implementation of a Signature Harvesting Scheme

In this section, we discuss the implementation of a signature collection scheme for signature pattern that suffers negligible local and global ambiguities.

Actually, the non-existence of global ambiguity allows forming a space-efficient algorithm for harvesting last touch signatures.

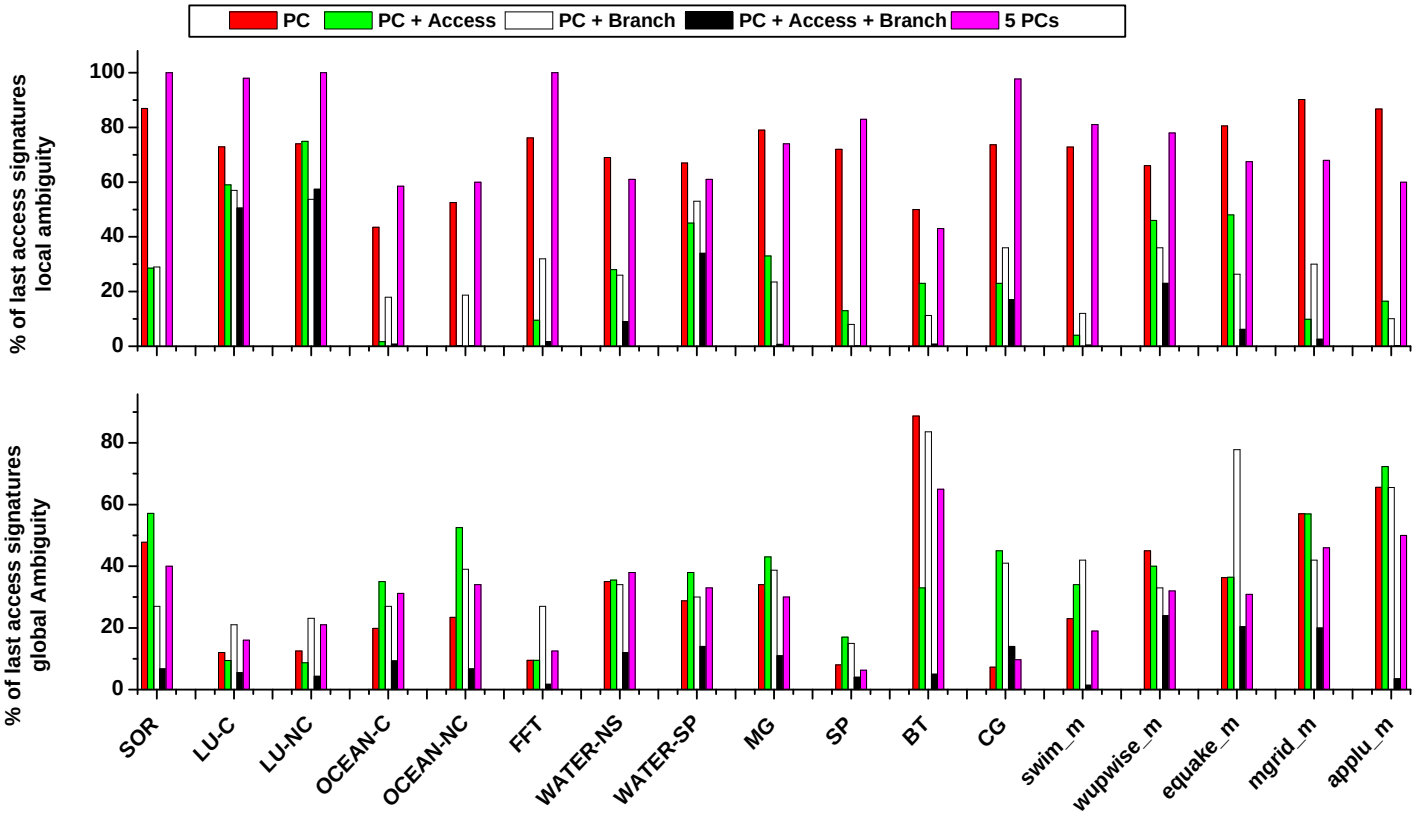


Figure 6.3: Last access ambiguity for different signatures.

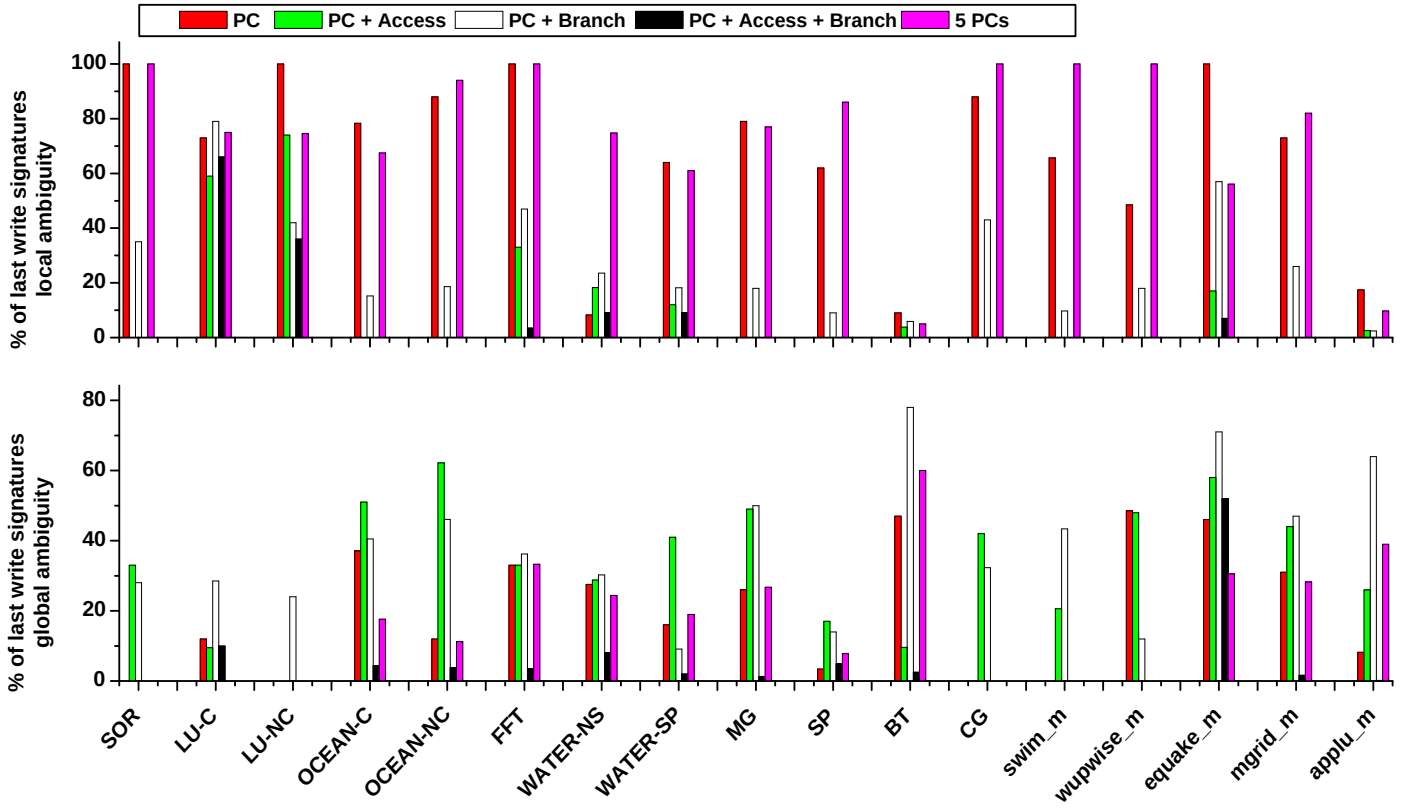


Figure 6.4: Last write ambiguity for different signatures.

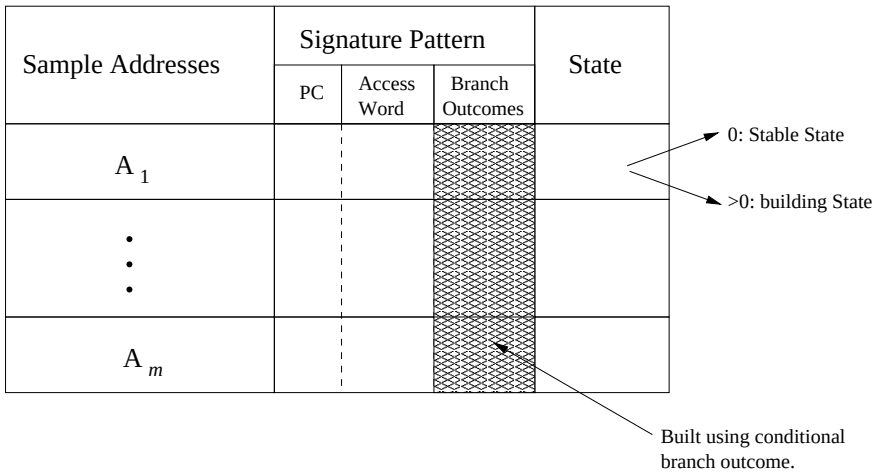


Figure 6.5: Signature collection tables. At any given moment, there are no two identical pattern with stable state.

The signature-harvesting table is composed of three fields, as shown in Figure 6.5. These fields are *address*, *pattern* and *state*. The address is a 32-bit field. The pattern under consideration is structured from PC, access word, and n branch outcomes after the access. Each cache line address requires at most 32-k-1 bits, where k is $\log_2$(bytes per cache line). As the signature is computed per session, the span of PC accessed is usually small. So, a small number of bits can be used for encoding. It is very common that a session (sequence of accesses between two barriers) exists within one routine with no intervening function calls. The *access word* requires k-2 bits. The *state* is composed of $\log_2(n)$ bits.

Each address has three stages during its lifetime. The first stage is *admission,* which happens when a memory address is accessed. The second stage is *buildup*, where the branch outcomes are collected. The third stage is *graduation*, which happens after n conditional branch outcomes are encountered.

The last touch signatures of a certain session are composed of the stable patterns remaining in the table at the end of a session.

The processing involved in these stages is as follows:

- *Admission stage*:

  When a new memory access is committed, the address is checked against already outstanding addresses. If an entry is not found, then a new entry is allocated and the PC and *access* fields of the pattern are initialized. The *state* field is initialized to the value of n. If an entry is found, then the PC and access field are re-initialized and the *state* field is reset to n.

- *Buildup stage*:

  On each conditional branch committed, addresses with *state* greater than zero are affected. A value of one for taken branch (or zero for not-taken) is shifted into the branch field of each signature. The *state* counter is decremented. If the *state* counter reach zero the address goes to *graduation* stage.

- *Graduation stage*:

  When the state field of a pattern reaches zero, the generated signature pattern is checked against other patterns with state zero. If the signature is matched, the address is evicted from the table as it does not represent a new signature. If the signature is not matched for any other address, then the entry is kept.

6.3.1.1 Convergence

The above algorithm provides a good coverage of the set of signatures that identify last touch/write. The conditions to be met are that the pattern structure does not suffer local or global ambiguities.

If addresses with state zero always hold a last touch signature, then any discarded entry should not represent any lost signature. If these signatures are not last touch signatures, then the discarded entry cannot hold a last touch signature. This assumes that this signature does not exhibit global ambiguity. A signature may be lost only if there is a common intermediate signature in the pattern of access. This is depicted in Figure 6.6. The problem may not be masked if all the following conditions persist:

- A common intermediate pattern exists in two access patterns that lead to a different last touch signature.

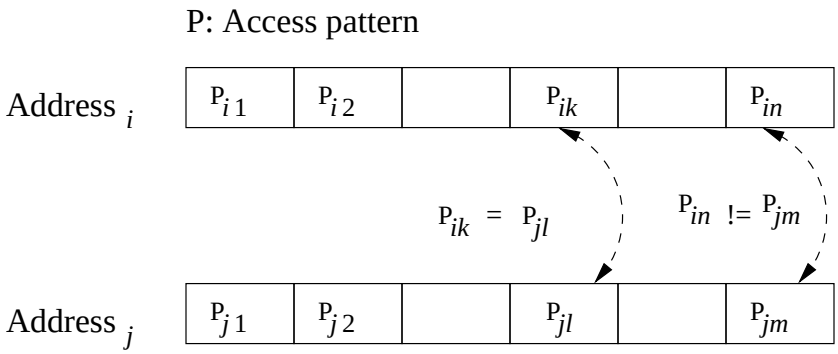


Figure 6.6: A difficult to collect signature scenario.

- This intermediate pattern is stable, i.e., n branch outcomes occur before the next access.
- Two addresses with these two signature patterns appear at the same time in the table at this intermediate state.
- For all addresses that have these two signatures, they reach this intermediate stable state in the table in the same order.

This unlikely scenario does not seem to hurt the prediction accuracy of the last-touch harvesting scheme as will be shown later. The likelihood of this scenario can be further diminished, if we increase the number of branch outcomes to be taken into account.

6.3.1.2 Accuracy of Signature Harvesting Scheme

In this setting, we test the ability of our algorithm to collect last touch/write signatures. We run each application on a four-CMP system. Prediction accuracy has three components: (1) percentage of signatures correctly predicted (compared to the actual signatures); (2) percentage of signatures not predicted; (3) percentage of signatures incorrectly predicted (relative to the total signatures). For all applications studied, the accuracy is almost 100%, as shown in Figure 6.7. The reason is that the signature selected suffers minor local and global ambiguity. This make the collection mechanism efficient in getting last touch signatures.

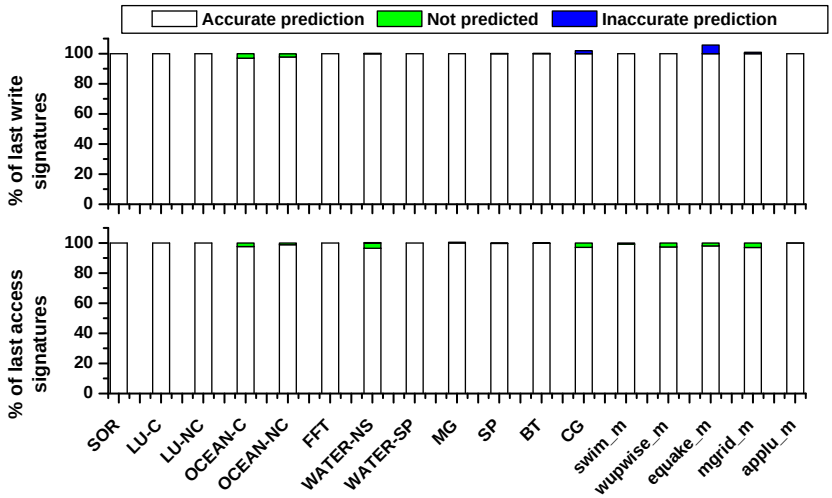


Figure 6.7: Accuracy of the harvesting scheme (for signatures collected by the R-stream).

6.3.1.3 Requirements of the Harvesting Algorithm

Table 6.3 gives the number of signatures for a given application, as well as the maximum number of intermediate outstanding entries at any given point of time, both for last write and last touch. It is noticeable that very few signatures are needed in most cases. The maximum intermediate states are slightly larger than the final state. The number of signatures to identify last write is much smaller than last access. As the operations required for collecting signatures are complex, then having a small number of entries is very desirable. To achieve good coverage, we suggest that the number of entries range from 64 to 128. To reduce the number of signatures, we can reduce the number of branches observed and increase access word granularity.

6.3.1.4 Accuracy of Slipstream Signature Collection

The same algorithm is run by the A-stream. We compare the accuracy of the signatures generated by the A-stream compared with those generated by the R-stream, presented in the Section 6.3.1.2. The matching accuracy of signatures is decomposed of accurately, inaccurately, and not predicted components. Figure 6.8 shows that signatures generated by the A-stream and the R-stream are almost 100% identical.

Table 6.3: Maximum number of signatures and the maximum count of intermediate signatures for (PC, access word, branch outcome) signature.

Benchmark	**All accesses**		**Writes only**	
	Outstanding	**Final**	**Outstanding**	**Final**
SOR	43	37	11	9
LU-C	28	24	19	15
LU-NC	25	21	15	11
OCEAN-C	217	206	108	106
OCEAN-NC	213	207	122	120
FFT	80	62	62	45
WATER-NS	50	50	31	28
WATER-SP	85	68	28	24
MG	128	114	44	43
SP	493	362	485	284
BT	871	772	634	603
CG	129	96	16	13
swim_m	131	125	92	90
wupwise_m	33	31	9	9
equake_m	767	766	352	310
mgrid_m	61	48	32	27
applu_m	806	414	681	676

This conforms with our claim of the identical behavior of both the A-stream and the R-stream.

6.3.1.5 Sensitivity to the Count of Branches

The number of branch outcomes used to identify the last touch prediction affects the requirements and the accuracy of the scheme. Figure 6.9 compares the ambiguity and requirements for last write considering 4 branch outcomes *vs.* 8 branch outcomes. The number of signatures is reduced by 30% on average for 4-branch signatures compared with 8-branch signatures. The ambiguity is increased by 3% on average for local ambiguity and 4% on average for global ambiguity.

The ambiguity is not changed very much for most applications. LU-C and WATER-(NS&SP) had a relatively high global ambiguity increase (15-22%). This confirms the intuition that branches help to reduce global ambiguity. It also indicates that the number of branches needed is application dependent.

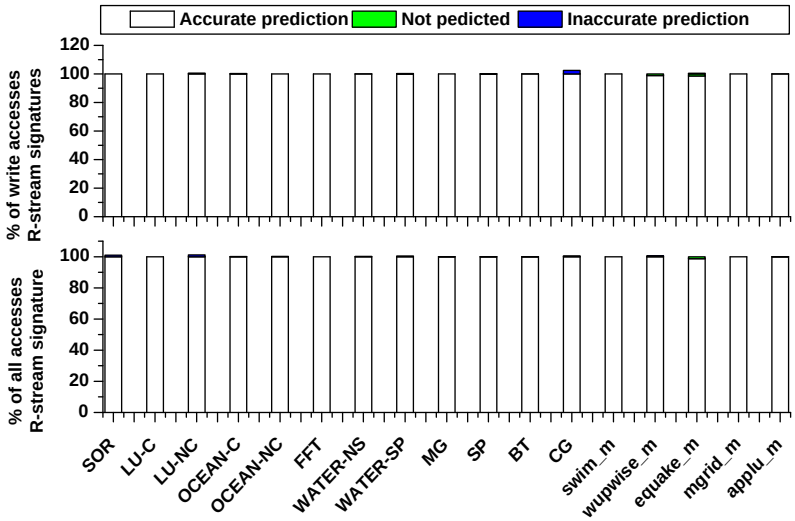


Figure 6.8: Matching accuracy of the A-stream collected signature (relative to signatures collected by the R-stream).

The reduction in signature count is important especially with applications with large number of signatures (for instance, OpenMP benchmarks.) Reducing global ambiguity is important as they lead to address independent signatures. Considering that the size of the table holding signatures should be fixed, we propose the following techniques to handle overflow:

- **Resource-driven adaptation**. Reduce the number of branches considered if an overflow occurs.

- **Redundancy elimination**. Some signatures may have complementary entries for branch outcomes. For instance, two signatures may have the same (PC, and access word) and complementary outcome for the ith branch. In this case, the two signatures can be replaced by one signature with ith branch set as do not care. This requires a three-value (two-bit) representation of each branch outcome.

- **Importance**. Signatures can be associated with different cache line count. We can use a counter associated with each entry in the table that represents the number of cache lines having this signature. This counter is incremented when

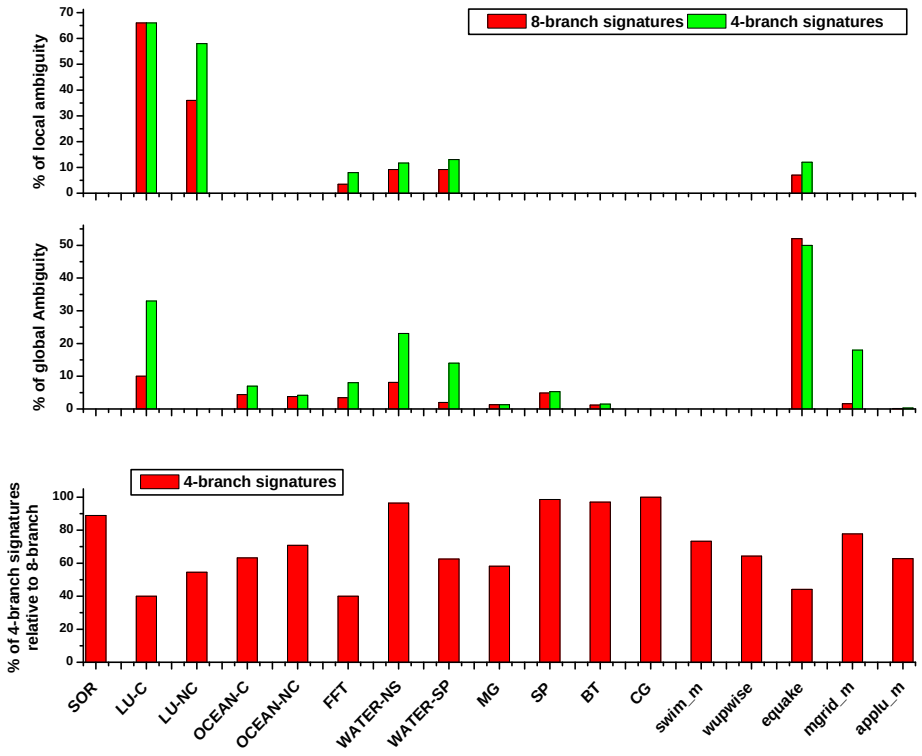


Figure 6.9: Sensitivity of (PC, access word, and branch outcomes) signature scheme to changing the number of branch outcomes.

a signature is repeated for two addresses and an entry is evicted. The signatures with lower count are candidates for eviction from the table when overflow occurs.

6.4 Counter Signature

The PC-based signatures discussed earlier are best suited for slipstream mode. They can also be adopted by a history-based scheme, if we add a per-session identifier (start session PC) and keep signatures across multiple sessions.

Slipstream execution mode enables a simpler scheme, based on the number of accesses to the cache line. As the A-stream and the R-stream have identical traces of

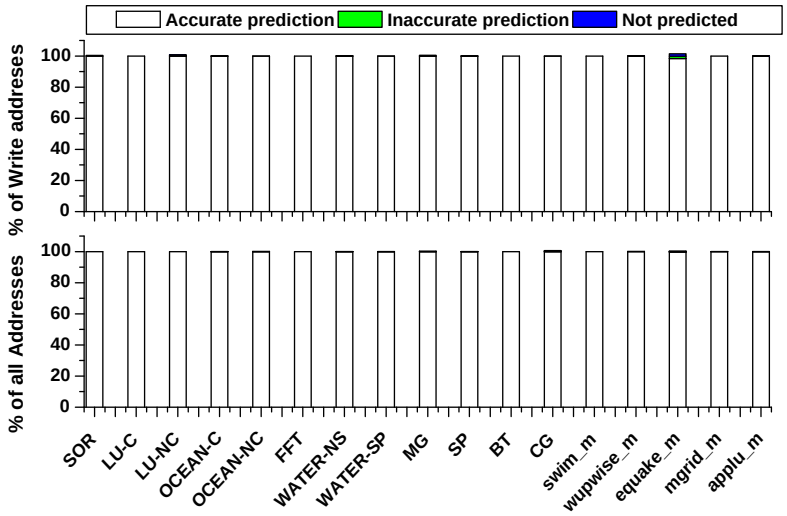


Figure 6.10: Matching accuracy between A-stream and R-stream for the counter signature.

accesses to shared variables, we can simply keep a counter per address that indicates the difference between the A-stream and R-stream accesses. A line is last touched when this counter holds a value of zero and the A-stream has reached the end of the session. This predictor does not suffer global ambiguity, as it is by definition an address-based predictor. Figure 6.10 shows matching accuracy of A-stream and R-stream number of accesses. The prediction accuracy is relative to the number of addresses accessed by the R-stream. Prediction accuracy has three components: 1) percentage of addresses correctly predicted (identical count produced by A-stream and R-stream) ; (2) percentage of addresses incorrectly predicted (different count reported by A-stream); (3) percentage of addresses not predicted (not accessed by A-stream). This figure shows the high match accuracy (almost 100%) between the A-stream and R-stream. Table 6.4 shows the average number of accesses per cache line within a session and the standard deviation of these accesses. As expected, monitoring only writes makes the number of bits required to implement this counter small. The accuracy is dependent on the number of bits chosen to represent the counter and also the application running.

The counter signature bits requirement can be further reduced by keeping track

Table 6.4: Counter signature average access count and standard deviation.

Benchmarks	**Average accesses count.**	**Accesses std. deviation**	**Average write accesses.**	**Writes std deviation.**
SOR	69	793	15.7	2
LU-C	589	644	245	40
LU-NC	589	644	245	40
OCEAN-C	34	57	12	4
OCEAN-NC	32	43	11	4
FFT	76	120	116	71
WATER-NS	1039	3035	30	56
WATER-SP	1112	2804	50	112
MG	93.8	280	13	4
SP	25	133	10	7
BT	74	0	30	52
CG	30	79	15	1.7
swim_m	46	38	16	2
wupwise_m	58	0	17	3
equake_m	61	368	33	23
mgrid_m	115	246	12	5
applu_m	86	578	25	95

of the number of cluster accesses, where each cluster is more than one access.

6.5 Temporal Locality and Access Clustering for Cache Lines

Temporal locality is usually taken care of while writing a parallel program. Temporal locality aims at minimizing cache misses through optimally reusing cache lines. Many high level optimizations such as tiling (or blocking) are applied on loops to achieve good temporal locality. This also makes LRU replacement policy an effective one.

This common style of writing programs (and compiler optimizations) makes most of the accesses to a cache line clustered. Last touch/write prediction can exploit this behavior to determine if the line is ready for a certain coherence action.

Based on generational behavior [22, 53], a certain period of time should elapse without accesses to the cache line to predict the line's readiness for coherence action.

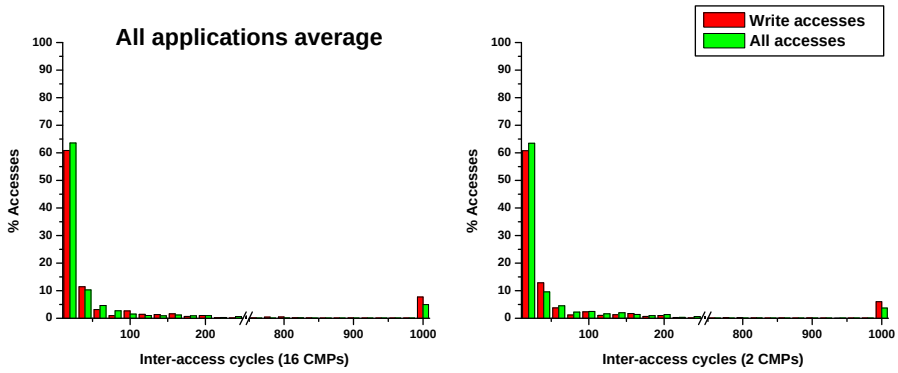


Figure 6.11: Combined distribution of inter access-time for all applications considered.

Figure 6.11 shows a combined distribution for all applications. It considers inter-access time for writes only and also for all accesses. The histograms are on multiple of 20 cycles. For each application, we studied the behavior for a system of 2 CMPs and a system of 16 CMPs. The applications were run in slipstream mode with global-one synchronization. As expected the accesses are very clustered. Figure 6.12 (page 90) shows histogram distribution of inter-access time for different applications. Table 6.5 shows the number of cycles needed to guarantee different percentages of coverage for inter-access time.

Our first observation is that clustering behavior does not change very much with the number of CMPs while it does vary for different applications. This indicates that program behavior and not system size dominates the factors affecting the clustering behavior. The second observation is that there is no general trend in clustering behavior of writes *vs.* all accesses. For instance, for SOR and OCEAN-C writes are slightly more clustered than all accesses. Conversely, for mgrid_m and applu_m all accesses are more clustered than write accesses. A noticeable proportion of the accesses are skewed from the general clustering behavior. Multiple factors contribute to this phenomenon: misses on intervening addresses, page faults, context switches, and other system services routines. Most importantly, increasing the needed coverage percentage un-proportionally increases the requirement in terms of the number of cy-

cles. This shows the only argument against this technique. To achieve high accuracy, we have to be overly conservative. We show clustering behavior for LU and OCEAN with two partitioning methods, contiguous and noncontiguous. Algorithms with contiguous partitioning tend to have more clustered accesses due to their efficient use of cache. MG (from Omni [1]) and mgrid_m (from SPEC_OMP2001 [3]) are OpenMP implementations based on the same serial code. They exhibit different clustering behavior. MG is rewritten and optimized by hand compared with mgrid_m.

Table 6.5: Number of cycles needed to achieve 80%, 90%, 95%, and 99% coverage of inter-access time.

benchmark	**Writes (2 CMPs)**				**All accesses (2 CMPs)**				**Writes (16 CMPs)**				**All accesses (16 CMPs)**			
	80%	**90%**	**95%**	**99%**	**80%**	**90%**	**95%**	**99%**	**80%**	**90%**	**95%**	**99%**	**80%**	**90%**	**95%**	**99%**
SOR	40	40	60	60	40	40	280		40	40	60	60	40	40	40	
LU-C	20	20	*		20	20	180		20	20			20	20		
LU-NC	20	40			20	40	220		20	40			20	40	320	
OCEAN-C	40	60	80	320	60	80	220		40	60	80	280	40	80	140	
OCEAN-NC	60				60	100			60				60	120		
FFT	40	60			40	60			40	60			40	40		
WATER-NS	100				20	60	340		60				20	80	360	
WATER-SP	40	800			20	60	240		40	720			20	60	240	
MG	160	200	200	440	180	240	260		160	200	200		180	260	440	
SP	800				360	800			900				400			
BT	140	300	560		120	200	420		160	460	800		120	200	660	
CG	100	100			80				100	100			80			
swim_m	360	460	600	800	120	320	460		140	180	260		80	140	200	
wupwise_m	20	60	60	220	20	100	280		20	60	60		20	100	200	
equake_m	40	620			60	380	640		40	460	880		40	200	500	
mgrid_m	600	640	660		260	560	640		640	660			280	580		
applu_m	120	380			60	100	340		180				60	140		

* Shaded cells indicate $\geq$ 1000 cycles.

This mechanism is much simpler than the signature mechanism, but it is more sensitive to runtime environment and compilation process. The program needs to be highly optimized to have good clustering. Clustering behavior is application dependent. If the optimization requires high prediction accuracy then a conservative (high) number of cycles should be considered as a decay period. The signature scheme proposed earlier (Section 6.3) can achieve high accuracy without being conservative (thus delaying optimization decisions).

6.6 Tracking Last Touch/Write for CMP

CMPs add another dimension to the problem of identifying the last touch or write as multiple processors may access the L2 cache line.

Shared memory optimizations require a unique per-node state, as any action on the cache line will affect the line state in all levels of the cache hierarchy. Having different levels of hierarchy requires communicating this state. This can be further complicated if the multiple processors share a certain cache level. For instance, tracking a last touch (especially reads) at the L2 can be complicated if the line is cached in all processors' L1 caches.

Tracking writes only is simpler as writes require exclusive ownership of the cache line. Writes by multiple processors require exclusive ownership and serialization of accesses. State related to writes can migrate along with the cache line to the processor that has the exclusive ownership.

Earlier discussion showed that identifying the last write is easier than identifying the last access. It also requires fewer resources for identification. On the other hand, some optimizations may have more system complexity if the last touch cannot be identified. For example, speculation after barrier synchronization may require identifying last touch/write to avoid misspeculation. If only the last write can be identified, then extra support is needed by the memory system, discussed in Chapter 7.

6.7 Identification of Lines to Be Accessed

Till now, we discussed identifying a last touch assuming that we know that the line will be touched. A simple scheme is to predict that all cached lines are to be accessed in every session, then last touch prediction scheme can be used to indicate readiness for optimization action. This approach is conservative and assumes incorrectly that all cached lines will be touched each session. A line may be cached but not touched at all within a certain session. If not touched, then this line should be treated as being last touched. The question that arises is how to distinguish between lines that will be touched and others.

Slipstream execution mode can provide a simple scheme to predict if a line will be touched (besides predicting last touch signature). The A-stream can mark accurately (and in advance) the lines that will be touched in a certain session, thus allowing other cache lines to be acted upon as soon as a coherence optimization is needed. It also defers lines that will be accessed to last touch/write prediction scheme.

6.8 Criticality of Misprediction

We define a misprediction to be critical if it leads to a loss of performance or has an irreversible effect. For instance, if a cache line is incorrectly indicated to be idle and ready for turning off to reduce power, then a later access to this line will suffer an unnecessary cache miss. This misprediction cannot be reversed without paying the cost of requesting this line from memory. This may not only degrade performance but also may waste more power in transactions needed to bring the cache line back from memory. A similar scenario can happen with an optimization like self-invalidation, except that only performance (and not power) will be hurt.

On the other hand, a prediction of line readiness for migration does not always trigger an action except when an external request for the data arrives at the cache. Mispredictions may be masked if no external event arrives to the cache. Marking a line for migration readiness can be reset upon receiving new accesses. Thus, this kind of misprediction exhibits less criticality to performance. If the readiness of a

Table 6.6: Summary for last touch identification schemes.

	PC, access, branches	**Count**	**Decay of access**
Concept	Use program behavior to determine address independent last access signature.	Monitor the access count for each address before last access.	Exploit temporal locality of accesses to determine last access (when a certain idle period elapses).
Accuracy	Given enough branches (8 or more), we can achieve high accuracy.	Given enough bits for counter (8 bits or more), we can achieve high accuracy.	Accuracy increases with the idle period.
Requirements	• Small number of last touch identifiers. • Complex circuitry.	• Counter associated with each cache line.	• Simple idle counter. • Logic to determine idle interval.
Pros	• PC-based signature scheme with small count of last touch signatures.	• Simple counting scheme.	• Very simple idle detection.
Cons	• Complexity of the logical circuit.	• Signature is address based. • Accuracy is dependent on the number of bits.	• One idle period for all cache lines. • High accuracy requires being conservative (very long idle period for most cache lines).

line for migration is used to support barrier speculation, then one misprediction can wreck multiple successful predictions. In this case, the criticality of misprediction is indirectly magnified.

Table 6.6 summaries the attributes of the major signature schemes, discussed in this chapter. It is notable that no single scheme combines accuracy, simplicity, and efficiency. The simplest scheme (based on decay of access) cannot achieve a high accuracy without being inefficient as it needs a large decay interval. On the other hand, the PC-based scheme (PC, access, branches) can achieve high accuracy along with efficiency at the cost of logical complexity. The counter-based scheme is very accurate, but it requires address-based storage of the touch count.

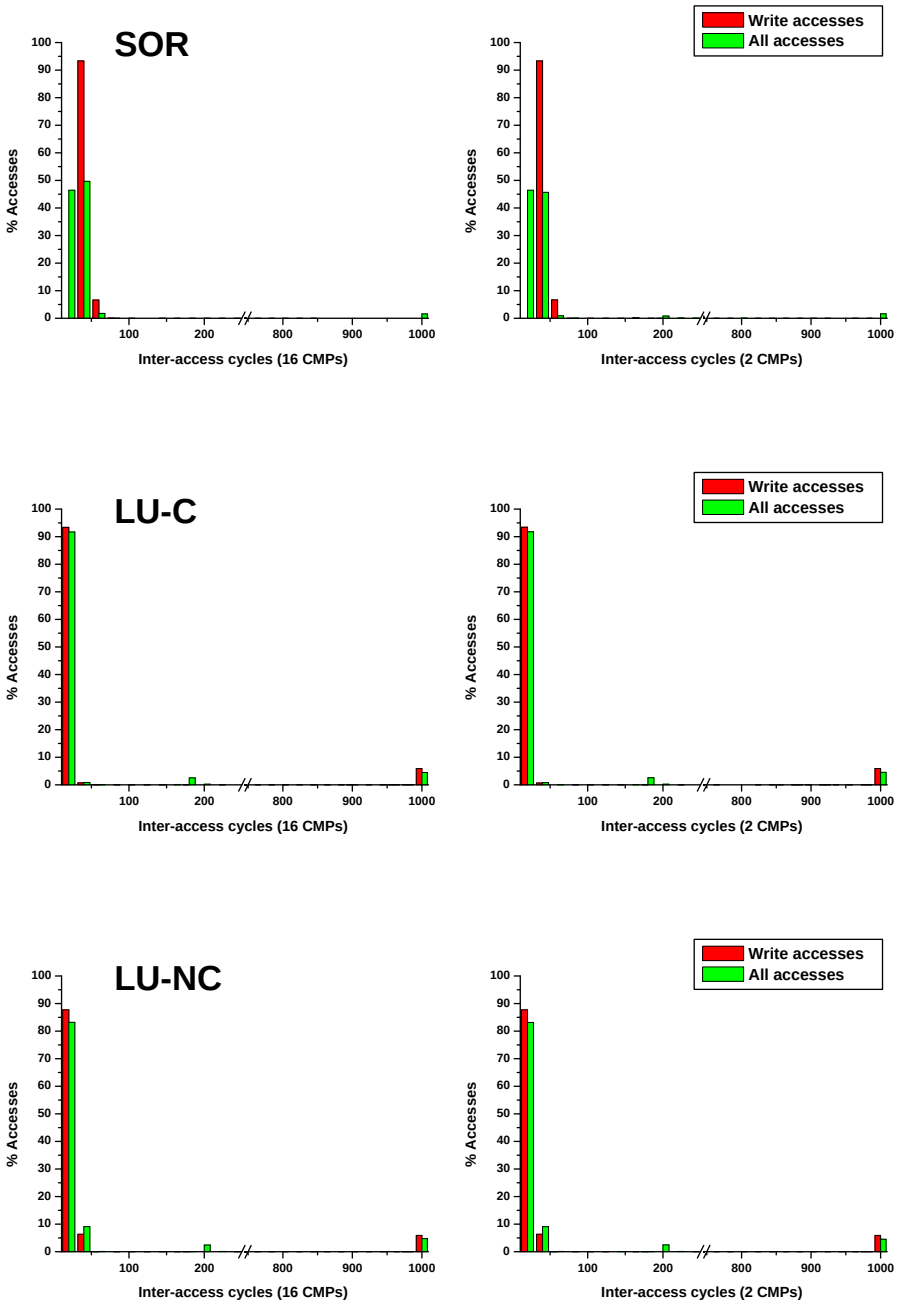


Figure 6.12: Distribution of inter-interval for writes and both reads and writes access for the cache block.

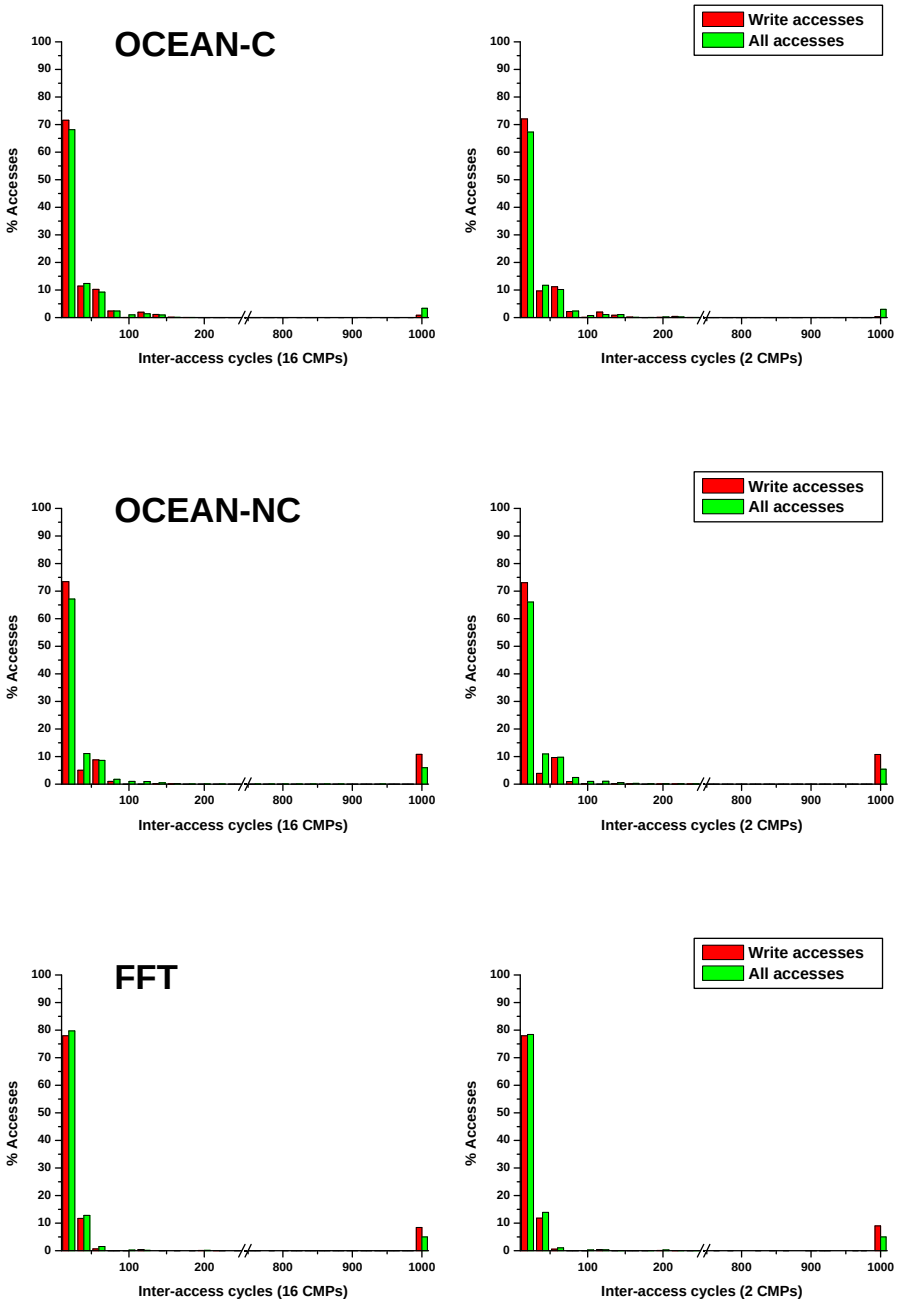


Figure 6.12: (*continued*) Distribution of inter-interval for writes and both reads and writes access for the cache block.

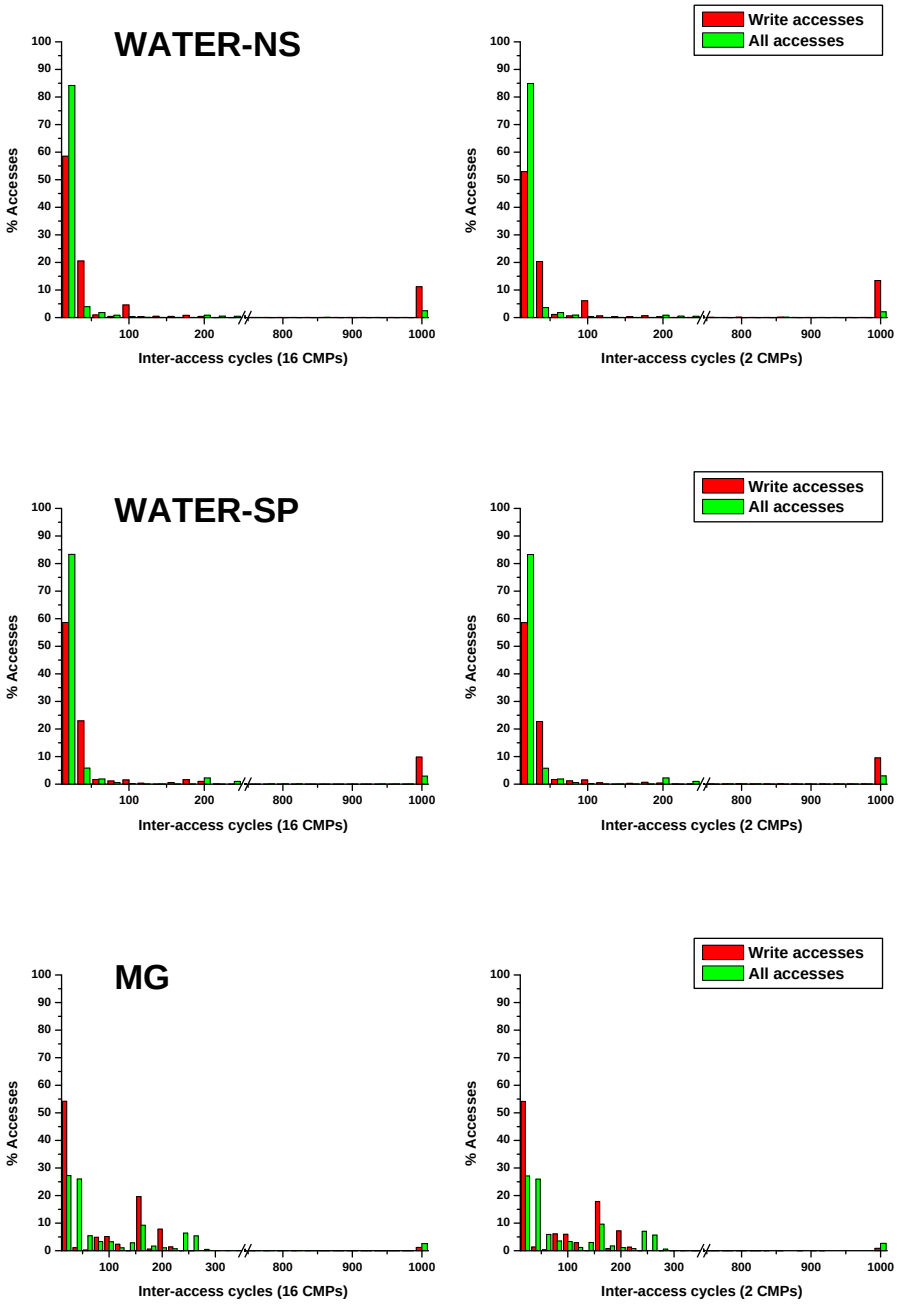


Figure 6.12: (*continued*) Distribution of inter-interval for writes and both reads and writes access for the cache block.

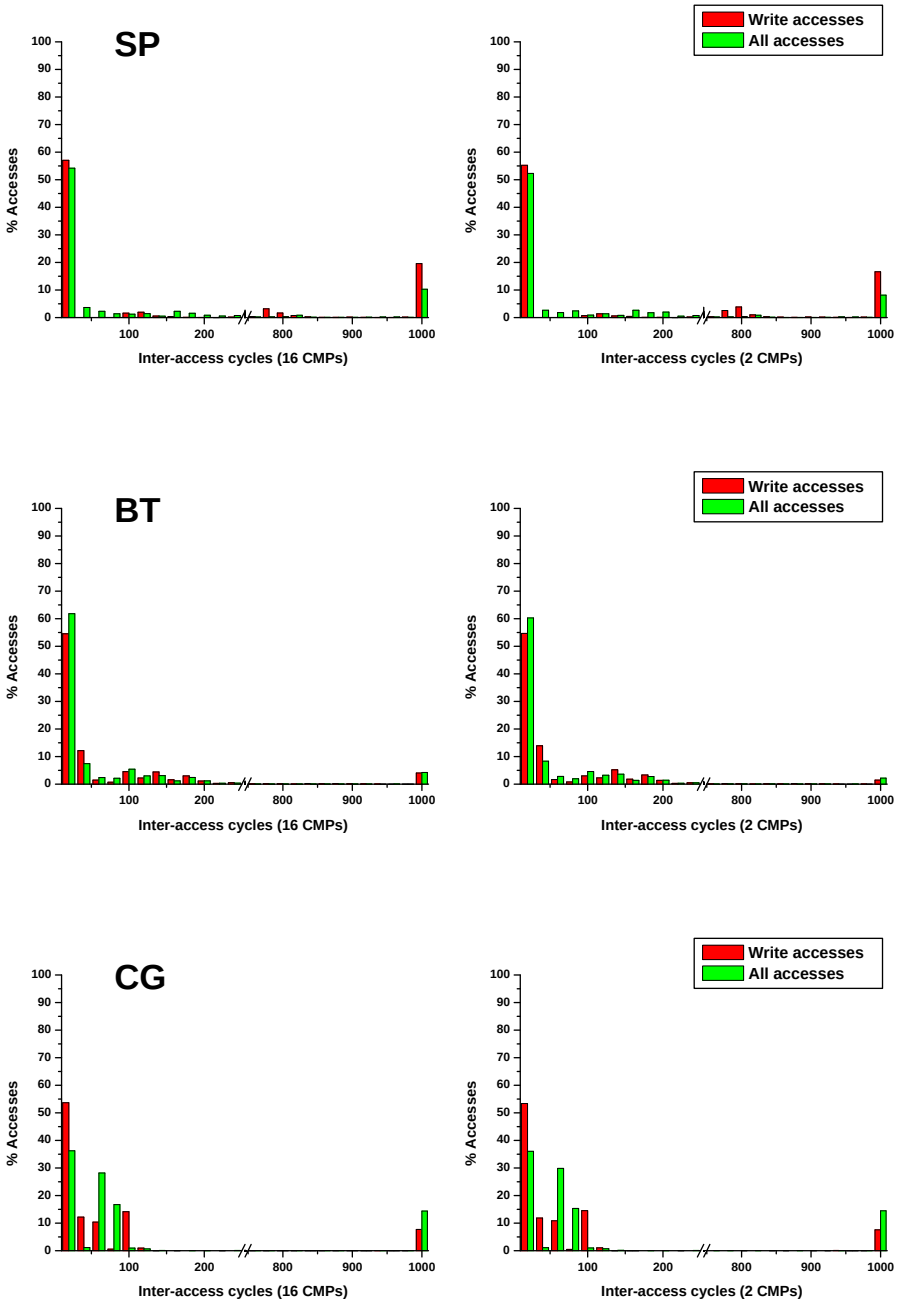


Figure 6.12: (*continued*) Distribution of inter-interval for writes and both reads and writes access for the cache block.

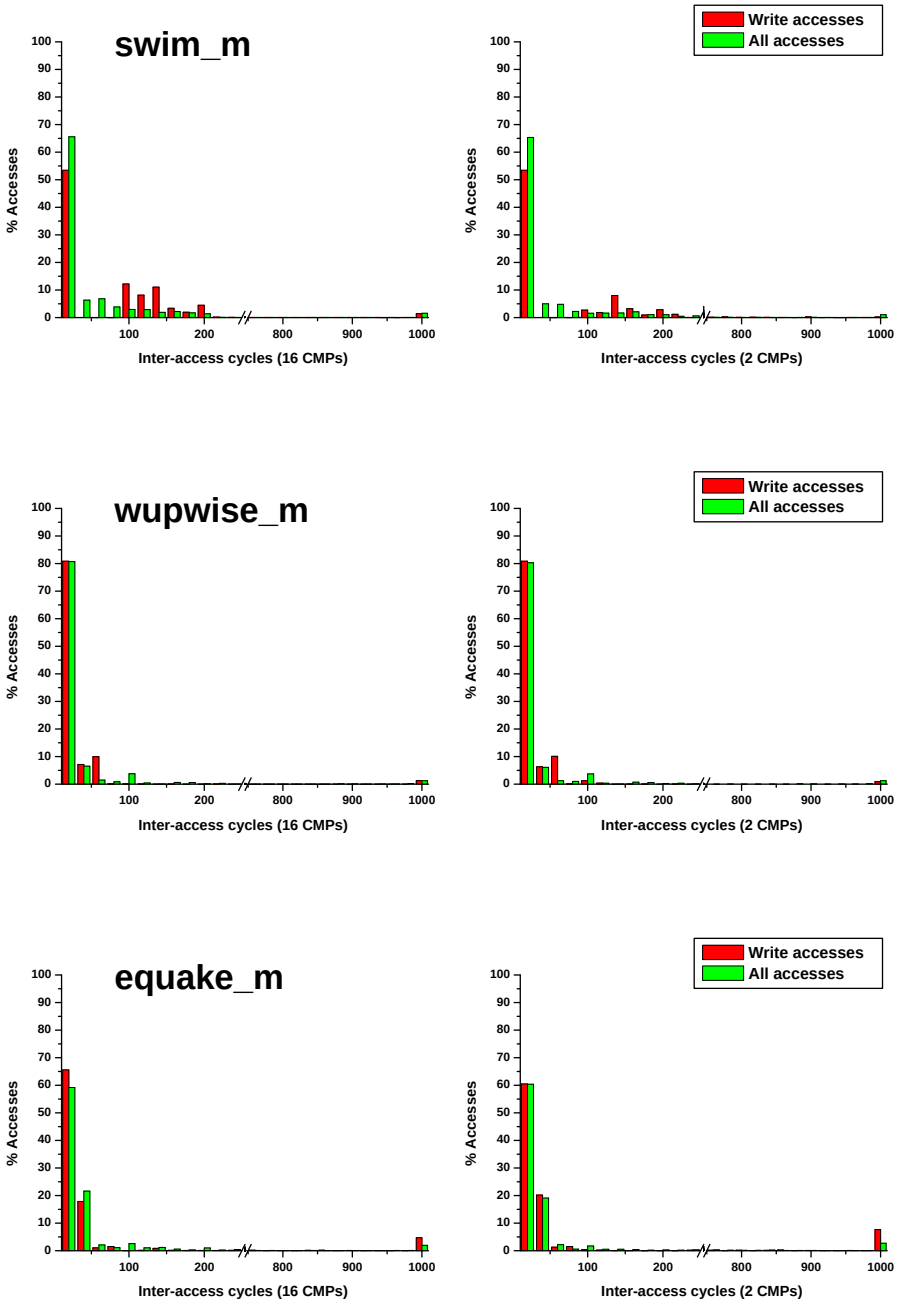
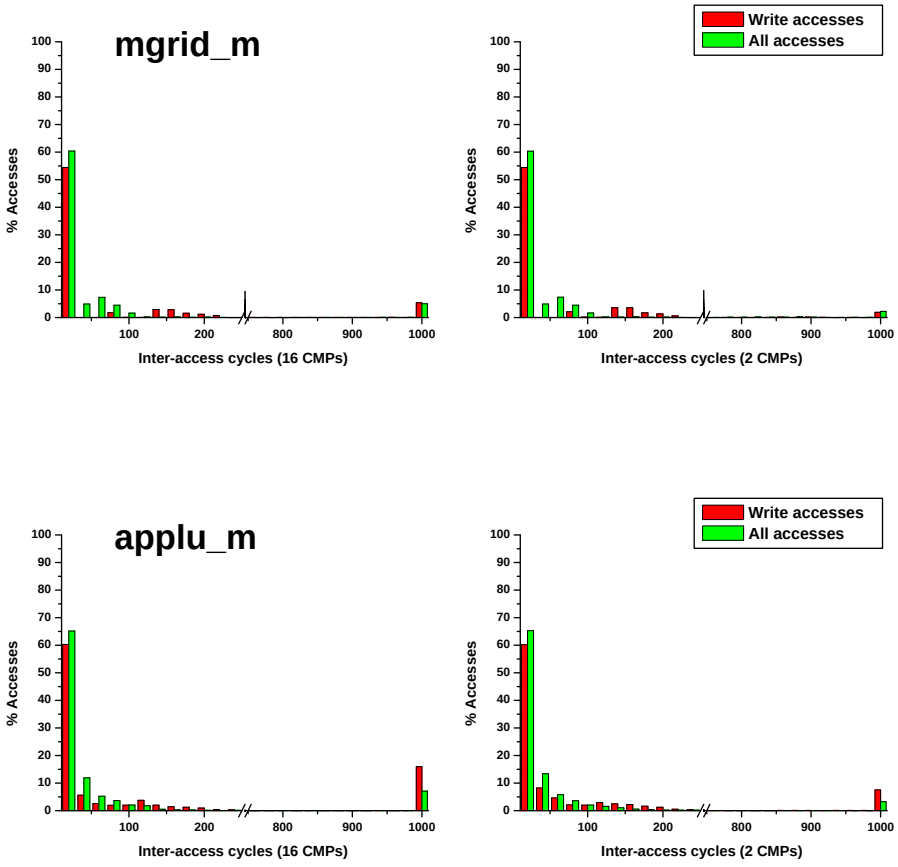


Figure 6.12: (*continued*) Distribution of inter-interval for writes and both reads and writes access for the cache block.



(a)

Figure 6.12: (*continued*) Distribution of inter-interval for writes and both reads and writes access for the cache block.

Chapter 7

Slipstream-based Synchronization Speculation

7.1 Barrier Synchronization

Synchronization primitives usually reflect some dependency constraints and sharing pattern characterization. For example, locks are best used to synchronize access to shared memory locations that are migratory shared. Locks guarantee mutual exclusion and organize handing off during migration. The order of execution and migration is not strict. Dependency is not strict in the sense that there is no necessary global necessary order of execution. The outcome exhibits an arbitrary execution order.

Flags are used to synchronize accesses in a producer-consumer relation. The dependency of the consumer process on the producer process contributes to the synchronization overhead. Flags represent a point-to-point synchronization primitive.

Barriers, on the other hand, are a bulk synchronization primitive. They synchronize many producer-consumer relations, effectively taking the place of many point-to-point synchronization events. The reason for using it is the ease of orchestrating programs. Barriers are also appropriate when a large data structure must be synchronized among multiple processes.

Though convenient for programming, the use of a barrier is a source of inefficiency in parallel programs. This inefficiency generally gets worse when we increase the

degree of parallelism.

One source of inefficiency is the overly restrictive data dependency imposed by the barrier, which associates many producer-consumer data dependencies with a single synchronization variable. The dependency relation represented by the barrier implies that all production on shared memory variables on all executing processes should be completed before any reference to these data after the barrier is allowed. The dependency effect can be reduced if the execution is balanced. In this case, all processes reach the barrier at the same time, thus minimizing the wait time. Load balancing (not execution balancing) is usually what is exposed to the programmer and is not an easy job in many cases. Load balancing does not necessarily lead to execution time balancing because of asynchronous operations. Therefore, it is difficult to completely remove the delay due to dependencies using a barrier.

Another source of inefficiency is the release overhead, recognizing when all processes have reached the barrier and then notifying all processes that this has occurred. Release overhead usually grows with the number of processes waiting on the barrier. It may be significant, especially in the case of having a non-broadcast medium like direct or indirect networks.

The importance of improving the performance of codes that intensively use barriers has grown with the emergence of OpenMP. OpenMP compilers usually insert barriers implicitly with many parallelization constructs. Data flow analysis is usually used to remove unnecessary barriers that do not protect any data dependency [39, 9]. These techniques usually face difficulties due to name aliasing and the unavailability of runtime information. Additionally, the compiler assumes the worst case scenario that conservatively inhibits removing barriers. In this study, we used an OpenMP compiler that eliminates redundant barrier synchronization when possible.

Complementary to barrier removal, speculation can hide part of the latency incurred by barrier synchronization [44, 17, 32]. This technique executes code protected by the synchronization primitive. Even in codes that are handwritten, where barriers are inserted only when necessary, there is still a chance for speculation. At runtime, execution can overlap such that no dependency violation occurs.

This optimization ignores the intended dependency imposed by the synchroniza-

tion. As the time to hide is usually large in case of barrier, many shared memory transactions are involved in speculation. Each access carries a possible violation chance. A single violation causes a rollback. Optimistic speculation after synchronization (assuming always no dependency) cannot only render the scheme useless, but also can harm performance. This is because of the added traffic/contention to the memory system, the premature migration of data, and the loss of part of cached data upon misspeculation and rollback.

In this chapter, we investigate techniques to reduce the chance of rollback. We advocate the need of confidence mechanism to help identifying when to speculate and when to delay/block speculation. We also aim at reducing the overheads associated with speculation after the barrier.

7.2 Barrier Speculation Overview

Barrier speculation aims at hiding barrier synchronization latency by executing code following the barrier. As the latency incurred due to synchronization is usually large, speculative state cannot be stored on the processor. Alternatively, speculative state is cached and the coherence protocol is used to detect dependency violations. The mechanism to support speculation is discussed in this section.

7.2.1 Speculation Initialization

The speculation can triggered using software [32] or hardware [44], depending on the implementation of the synchronization routine. If the synchronization is in hardware, then the speculating process takes over the processing resources leaving the synchronization for the synchronization circuitry.

If the synchronization is in software, then a special unit is needed to execute the synchronization routine while the processing unit is executing in speculative mode. This approach is adopted in this work, as software synchronization is more common for CC-NUMA systems.

When speculation begins, the processor state must be checkpointed. This check-

pointed state is used to recover the processor state if the speculation process fails.

The barrier routine used on this study is a tree-barrier that is flag-based. Figure 7.1 shows the data structure and software code used to implement this routine. For clarity, the code shows an implementation that only supports 2^n processes.

The barrier routine is composed of two phases. The first phase marks the arrival of a process to the barrier routine. The second phase is waiting for a signal that all processes reaches the barrier. A notable difference of this routine from earlier proposals [32] (shown in Figure 7.2) is that this routine does not use locks to manage the first phase of the barrier. This allows overlapping the whole barrier routine with speculation. Martinez [32] uses a routine that serializes the first phase of barrier using critical section and overlaps speculation with only the second phase of barrier. This serialization of the first phase by itself creates an artificial load imbalance. In case all processes reach the barrier at the same time, they will serially execute the first phase. Those processes that enter the barrier first may go into a deep speculation (which is unsafe) unnecessarily. To mark the arrival of all processes, the tree barrier implementation has $O(\log_2 n)$ memory transactions in the critical path executed non-critically (without locks) *vs.* O(n) critically executed in the lock-based scheme.

To overlap the execution of the barrier synchronization code with the speculative code, we require the barrier routine to be a leaf function (that has no calls to other routines). This includes system or synchronization routines (like locks). The variables count should be small enough to be held in registers. Compilers treat leaf functions with small storage differently. They make the argument passed and variables allocated in registers. These conditions allow the barrier checker code to be independent of stack (which can be modified by the speculative code). The structure holding barrier variables should be aligned in one memory page, thus one virtual to physical mapping is enough. These restrictions allow execution of the barrier checker slice of code in parallel (by an independent thread) with the speculative code without interference.

```
struct barrier_struct {
    volatile int barrier_sense;
    volatile struct {
        int _v;
        char _padding[CACHE_LINE_SIZE-sizeof(int)];
    } barrier_flags[PROC_COUNT];
};

void tree_barrier(struct barrier_struct * barr,
                  int proc_id, int nproc)
{
    AUG_ENTER_KERNEL;
    /* branch and fork */
    if(BDOOR_SPEC_BARR) {
        sen0 = ! barr->barrier_sense;
        Mask = 1;
        src = proc_id ^ Mask;
        while(src<proc_id) {
            while(barr->barrier_flags[src]._v != sen0) ;
            Mask = Mask << 1;
            src = proc_id ^ Mask;
        }
        if(proc_id == (nproc-1) )
            barr->barrier_sense = sen0;
        else {
            barr->barrier_flags[proc_id]._v = sen0;
            while(barr->barrier_sense != sen0) ;
        }
    }
    else {
            /* Speculation is triggered here. */
            /* We have to store a recovery point. */
            AUG_BARRIER_START_SPEC;
            return;
    }
    AUG_EXIT_KERNEL;
    AUG_BARRIER_END_SPEC;
}
```

Figure 7.1: Speculative tree barrier structure used in this study.

```
struct barrier_struct {
  volatile int sense[PROC_COUNT];
  volatile int global_sense;
  int Count;
  volatile int lock;
};

void lock_based_barrier(struct barrier_struct * barr,
                                    int proc_id, int nproc)
{

    barr->sense[proc_id] = !barr->sense[proc_id];
    /* Block speculation if the speculation unit is busy */
    /* This prevents nested speculation  */
    SS_EXPOSE;
    LOCK(barr->lock);
    barr->count ++;
    if(barr->count == PROC_COUNT) {
        barr->count = 0;
        barr->global_sense = barr->sense[proc_id];
        UNLOCK(barr->lock);
    }
    else {
        UNLOCK(barr->lock);
        SS_SPIN(barr->global_sense, barr->sense[proc_id]);
    }

}
```

Figure 7.2: Lock-based speculative barrier [32].

7.2.2 Managing Speculative State

Speculative state is usually maintained at the cache and is not allowed to propagate further. As will be discussed later, forcing speculative state out of the cache system is a violation that requires a rollback.

At least one bit is needed to mark those cache lines that were touched during the speculative session. We may need a marker bit on the MHT, if we allow a mixture of speculative and non-speculative requests in flight. This is the approach adopted in this study. During switching from speculative to non-speculative state, some MHT entries need to be repaired (i.e. changed to non-speculative). An alternative solution

is to use memory barrier instructions during switching between speculative and non-speculative phases of execution. This approach is simpler but it adds latency to the switching process.

The speculative bit is usually associated with the lowest level of cache (2^{nd} level, in our case). This helps detecting violations of speculation upon external events, as will be discussed later in detail. We may need a speculative bit in the higher level of cache to avoid propagating speculative touches on every access to the lower level. A speculative touch is propagated to the lower level only when we do not have the speculative bit set on the higher level. If the line is brought to the higher level while it is speculative, then the speculative bit in the higher level is set as it arrives to the cache.

7.2.3 Speculation Violations and Rollbacks

Speculation violation is detected using the coherence protocol. The violation is detected in two forms: (1) invalidation of a cache line that is speculatively read or written, or (2) an update request for a line speculatively modified.

Rollback involves invalidating cache lines that were speculatively modified. The speculative copies are not propagated in this case to the memory. Cache lines that are speculatively read are just unmarked (the speculative bit is reset).

The processor state should be restored from the checkpointed state. We have to make sure that the process active on the processor is the one that started speculation. We also have to make sure that the process is in the user mode (*vs.* kernel mode). If not, then the restore process should be delayed until these conditions are met.

7.2.4 Speculation Success and Committing Speculative State

When the last process reaches the barrier routine, then all processes start exiting the barrier. On their way out, each triggers a termination of successful speculation. This termination promotes the speculative state of the speculating process to be non-speculating process (i.e. speculative registers are committed). The cache lines marked speculative are unmarked (reset).

Speculation success usually involves small overhead and can be performed in parallel for all cache lines.

7.3 Barrier Speculation Overheads

Speculation after barrier synchronization involves additional overhead on the system. It usually involves more memory transactions during data access, rollback, and to acquire safe system state. These overheads are discussed in detail, in this section.

7.3.1 Memory Updates

Speculation involves updating cache lines. As speculation may fail, a recovery copy should exist. As discussed earlier the speculative copies are cached and do not propagate in speculative state to memory. Data modified speculatively will therefore have a backup in the memory. If the data to be modified speculatively is dirty in the cache, then we need to send a copy to the memory before starting the modification. This kind of transaction is not supported by most directory protocols. Commonly, directory protocols support replacement hints and/or writebacks. Both of these transactions imply losing the cached copy, and the directory removes the sender from the sharer list.

The cost of the update transactions are protocol dependent. Following the guideline of DASH [28], the update transactions do not require an ACK from the directory. Following the guideline of Origin 2000 [26], the update transaction requires an ACK. This makes this transaction expensive and thus delays speculation whenever it encounters this scenario.

7.3.2 Rollback and Flushing Corrupted Lines

Rollbacks involve getting rid of lines modified speculatively. In Origin 2000 protocol, the replacement of cache line does not involve a memory transaction. In DASH protocol, replacement involves a memory transaction (a replacement hint).

If the transaction that causes rollback is due to an external request affecting a speculatively modified cache line, then the external request may be NACKed. This is necessary if there is an update in flight to the directory due to speculation. This update carries the only valid copy in the system. In networks that do not guarantee point-to-point ordering, the update may be delayed. In Origin 2000, updates are tracked in the cache, so NACKing can be applied only if there is an update in flight. For DASH, the updates are not tracked in the cache, so we need to NACK for all speculatively modified lines.

From the above discussion, we see that one scheme (DASH) can perform speculation operations more cheaply, while the other (Origin 2000) performs the rollback more cheaply. As speculation usually involve both scenarios, it not obvious which is preferable.

The protocol adopted by SimOS follows the DASH guidelines regarding this issue. One argument favoring this approach is that with a confidence mechanism that make most of the speculation attempts successful, we better optimize for successful speculation cases.

7.3.3 Lazy Context Switch of Coprocessor Registers and Speculation

Lazy context switching aims at minimizing the overhead of saving and restoring coprocessor registers during a context switch. This optimization can complicate the speculation mechanism, as the registers of coprocessors cannot be assumed to be valid all the time. In this section, we review the lazy context switching concept and its interaction with speculation. We also propose some workarounds to solve this problem.

7.3.3.1 Overview of Lazy Context Switch

Coprocessor hardware (for example FPU) typically has a single set of hardware registers. To use these registers by multiple processes, the operating system reserves a special memory to hold each process's registers while not executing. Unlike normal

processor registers, the operating system tries to avoid loading and saving coprocessor registers upon each process switch. This is because switching consumes a significant amount of CPU cycles.

Modern CPUs provide a mechanism to support this behavior. A usability bit is maintained for each coprocessor. This usability bit indicates the state of the coprocessor registers as usable or unusable. Initially, this bit is set unusable. When the program first executes an instruction that uses these registers, an exception is raised and the operating system initializes/restores the coprocessor registers before resuming this instruction. The operating system maintains a variable indicating which process has the ownership of the registers. The coprocessor registers are stored only when a process other than the owner tries to access the coprocessor registers. They are loaded back only when processor tries to access them while not maintaining the ownership.

This causes a large savings, as many programs do not use these special registers.

7.3.3.2 Lazy Context Switch and Speculation Mechanism

Almost all scientific workloads use FPU registers. Speculation requires checkpointing registers of the processor and/or coprocessor to recover in case of misspeculation. The checkpointed registers should be a valid state of the registers. With lazy switching a valid state of coprocessor registers is not guaranteed all the time.

Another problem may happen during restoration of checkpointed register state in case of rollback. The OS should see the recovering process as an owner of the FPU registers. This is not also guaranteed under lazy context switching.

7.3.3.3 Proposed Solutions

The discussion above shows the need to have a valid state during both checkpointing and recovery. A safe solution is to inject instructions that trigger the OS to turn on the usability bit both before checkpointing and before recovery. This solution can be expensive (as it may involve multiple memory accesses) for checkpointing and recovery. This approach is adopted in this work.

The overheads of these operations can be reduced, if we detect sessions that have

no access to coprocessor registers and waive injecting these special instructions for them.

A naive alternative is to rollback any time the processor needs to handle an interrupt (which may lead to a context switch). This approach is the simplest to implement but can be very destructive and thus lead to performance degradation.

Another approach is to adopt eager updates during switching when running with speculation allowed. This requires the OS to provide this facility. This may also incur high overhead.

7.3.4 Dependency Violation Problems

Rollbacks can be caused not only by true dependency violations but also by anti-dependency, output-dependency, and multiple speculators. One of the main sources of these violations is false sharing due to cache line granularity. Barrier speculations usually do not involve versioning. That is why we cannot support different speculative copies of the same cache line co-existing.

These dependency violations can adversely affect performance even if they are infrequent. The reason is that they affect other work that did not involve any violation. As discussed earlier, rollback involves evicting all lines speculatively modified, whether they were cached before the start of speculation or brought to the cache during speculation. These evictions involve lines that did not suffer any violation of dependency. This can be contrasted with prefetching and some of the relaxed memory consistency models. In the latter case, only transactions that cause violations are affected, and other successful transactions are not affected.

In summary, unsuccessful transactions can wipe out the effect of many successful transactions. Not only that, they also turn successful transactions into harmful ones. These successful transactions (that are eventually invalidated) have consumed bandwidth, contend resources to succeed, or their data may have remained in cache if no speculation were allowed. Good examples of these victim successful transactions are transactions with private variables, for example stack variables.

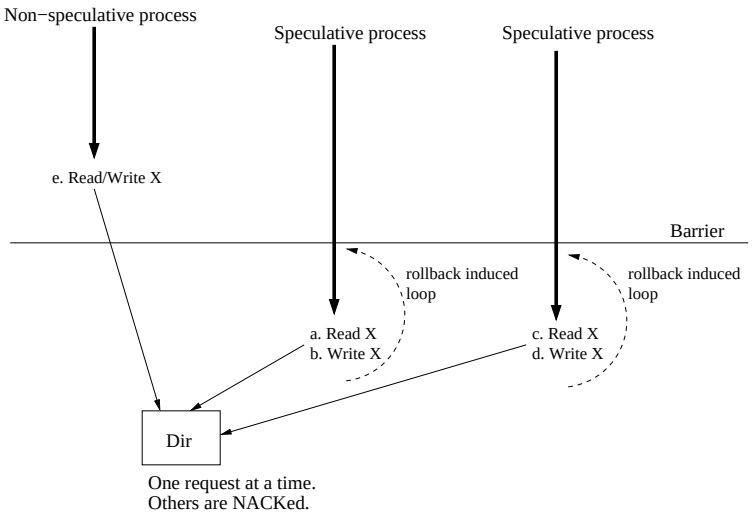


Figure 7.3: Live-lock due to barrier speculation.

7.3.5 Livelock Under Speculative Execution

Most CC-NUMA protocols use directories to manage coherence. These protocols do not guarantee fairness. They try to avoid deadlock by not buffering unsuccessful transactions. Instead, they NACK requesters and ask them to retry later. Allowing a failing speculator to retry indefinitely causes starvation for non-speculators.

Consider the situation depicted in Figure 7.3. Multiple speculator processes try to access a cache line that is also to be accessed by a non-speculator. The speculation and rollback mechanisms create a dynamic instruction loop that is not intended by the original program. In this case, it is possible for the speculators to keep the pending bit busy by the retrying failing speculator. The execution sequence (a→b→c→d)* is a possibility. This could potentially make the non-speculator (e) block indefinitely.

Later on in this chapter we will discuss techniques to avoid this scenario.

7.3.6 Rollback Cost

We discussed earlier the effect of rollback, and we showed that it does not have a neutral effect on performance. In summary, the reasons that make rollback have an adverse effect on performance include:

- Losing cached data that was written (hurting the speculator in the future).
- Losing all data written that was prefetched from memory (excess traffic).
- Unnecessarily updating data that are speculatively written if they were originally cached dirty (excess traffic).
- Hurting the non-speculator critical path due to
    1. Premature migration of data.
    2. Unnecessary contention for resources.

For these reason, there is a need to identify transactions that cause misspeculation and to block/delay speculator from performing them. We will discuss, in the next section, techniques to assign confidence to speculation to avoid rollbacks.

7.3.7 Reducing Speculation Overheads

Speculation overheads can be reduced through avoiding unneeded memory updates. Earlier proposals [44, 32] used a single bit to indicate speculative touch. A speculative read is not distinguished from a speculative write. Given the earlier discussion of the speculation cost, we find that there is an opportunity to save memory transactions in the case of speculatively reading a dirty line in the cache. With only one bit supported, this cache line must be updated to the memory before accessing the cache line.

Using two bits for speculation, we can distinguish a speculative read from a speculative write. In this case, a speculative read to a dirty line will not require updating the memory. This can have a good payoff, not only in reducing the update frequency to the memory, but also in avoiding some of the rollback scenarios discussed in Section 7.4.2.

Actually, we will see that a significant percentage of memory updates can be avoided with the proposed extra bit.

7.4 Confident Barrier Speculation and Slipstream Support

The objective of this section is to discuss techniques used to reduce the chance of rollbacks as they adversely affect the performance. These techniques try to filter out dangerous speculation scenarios. Speculation blocking is usually associated with resource problem, such as cache overflow and entering a kernel routine. We extend blocking to include scenarios that can lead to violation of dependency. The blocking may be temporary (until data is ready for migration) or until the end of barrier. We also support deeper speculation through breaking anti-dependence using a memory renaming technique.

7.4.1 Speculation Blocking

Speculation after barrier synchronization can be blocked in different cases. The reason for blocking can be due to resource problems or a high possibility of speculation failure. In this case, the synchronization code still executes in its special unit. When we reach end of barrier speculation, the speculative state is promoted to become the process state. Note that blocking speculation does not imply committing speculative state, as discussed in Section 7.2.4.

7.4.1.1 Cache Overflow Blocking

As discussed earlier, speculatively modified cache lines should be kept in cache and should not be replaced. This necessitates modifying the cache replacement policy (usually least recently used (LRU)) to avoid replacing these lines. If the whole set is occupied with speculative touched lines and the speculative process tries to replace one of the cache lines, then it is better to stop speculation and save the work already done.

The above approach was adopted in earlier proposals [44, 32]. In some scenarios, a replacement is forced if the replacement is requested by a system routine, since blocking a system routine can freeze the whole system. The speculative process can

trigger these system routines, for instance, to handle a TLB miss or a page fault. In this case, we have to do the replacement and force a rollback.

In our implementation, we limited the number of speculative cache lines in an n-way set associative cache to be n-1. At any given point of time, at least one entry is available for replacement.

As we discussed earlier, avoiding rollback is an important objective even at the expense of limiting the available resources for speculation.

7.4.1.2 Kernel Enter Blocking

If the speculating process tries to enter a kernel routine, then we block speculation (i.e. we freeze speculation up to this point). By kernel routines, we mean I/O routines, synchronization routines, and scheduling routines. The problem of entering these routines while in speculative state is that the effect is irreversible, or the effect is unpredictable. I/O routines have irreversible effects and should not be executed. Executing a scheduling code, especially with dynamic scheduling, has unpredictable results. The speculating process can go astray due to this speculative execution. Although the rollback mechanism provides a way to recover from the second scenario, it is still advisable to avoid these cases if possible.

The above discussion shows the need to annotate some of the system and parallelization libraries to indicate where speculation should be stopped.

7.4.1.3 Maximum Speculation Attempts Blocking

The number of speculation attempts should be limited. This will give a safeguard if other protection mechanisms fail. If the number of speculation attempts is exceeded, then we block speculation, i.e. we expose the synchronization routine. In this study, we limited the number of speculation attempts to five. We also insert a fixed[1] retry delay after rollback. This can reduce the chance of races between speculator processors and non-speculators.

[1] A random retry delay would be better.

7.4.1.4 Non-speculative Barrier

In addition to the above blocking scenarios, speculation should not be allowed if the barrier routine is used for internal management. For instance, in OpenMP, some barriers are used to prepare a new scheduling decision. They may be also used to synchronize slaves until a new job is available. Speculation after these type of barriers leads to having the speculator execute unpredictable codes. They may also raise exceptions that usually trigger rollbacks.

In our implementation, we have added an argument to barriers that specify if we can speculate after this barrier or not. The differentiation between these two types of barriers is easier in the case of OpenMP, as the compiler is responsible for most barrier insertions. The scheduling barriers are usually internally managed.

7.4.2 Avoiding False Rollback

A false rollback is caused by multiple speculators that have conflicting requests to the same cache line. In Section 7.3.5, we discussed that this can cause livelock.

In our scheme, we adopted a simple technique to solve this problem. First, any speculator should flavor any request going to the memory as a speculative request. If the directory needs to communicate with another cache to complete the memory transaction (for example, requesting an invalidation), then it sends the request flavored to indicate that it is from a speculative requester. At the receiving node, if the line is non-speculative, then the request is handled normally. If the line is speculatively touched, and processing the request will cause a rollback, then the request is NACKed. The reply is also flavored to indicate that the NACK is due to another speculator. The directory in return NACKs the requester indicating that another speculator has the line in speculative state.

The sender process in this case should block the speculation, as discussed in Section 7.4.1. Flavoring the NACK helps to determine the type of the action at the requester. Normal NACKs do not trigger blocking, but this flavored request/reply scenario can trigger blocking.

There is a scenario where a speculative request can succeed on another speculative

line. This happens when the requester tries to read a line speculatively, while it is dirty in another cache but was speculatively read (and not written). In this case, the receiver of the request can send an update to the memory and lose exclusive ownership, while not violating the speculative state in the cache. This shows another scenario where differentiating between speculative read and write is necessary.

Interestingly, this simple addition to the coherence protocol is one of the most effective optimizations to avoid rollbacks, as will be shown later.

7.4.3 Avoiding Anti-dependence Rollbacks

Dependency rollbacks can be avoided in some cases. In this section we discuss how to avoid rollback due to a non-speculative read while the line is speculatively modified. This technique exploits the availability of a clean copy of the cache line while it is being speculatively modified in a cache. This optimization is proposed by Sato et al. [44] in a bus based system. The main idea is that if a non-speculator asked for a cache line while a speculator is exclusively holding it, then the memory system, in response to this request, can supply the copy available in the memory. The memory line supplied is marked as an *expiring copy*, which means that this line should be evicted when we reach the next barrier. This technique resembles register renaming to remove anti-dependence in an out-of-order core.

To support this optimization in a CC-NUMA system, we need to distinguish lines with speculative exclusive ownership from normal exclusive ownership. This requires an additional bit at the directory.

Speculative exclusive ownership requires a memory transaction in all cases. If the line is not cached, then it is requested form the memory. In this case, a flavored request can indicate that the request is from a speculative process. If the line is exclusively cached, then the line should be updated to the memory in a special transaction that was discussed earlier. In this case, the line can be marked as speculative exclusive. When a non-speculator requests this line for a read, then the line is sent with the copy in memory and an indication that this line should expire at the next barrier. When the barrier routine is reached, expiring copies should be invalidated. The directory

does not keep track of expiring copies in the system. If a non-speculator requests this cache exclusively, then an invalidation is sent to the speculator. Additionally, messages are sent to all nodes to evict this line if an expiring copy exists. This is the approach adopted in this work. We favor this technique, as we assume that this scenario will happen infrequently, especially with dependency checking discussed in Section 7.4.5.

An alternative approach is to use a speculative node id (instead of one bit) and to use the sharer list to keep track of expiring copies. This enables saving many messages in the case of exclusive request by a non-speculator.

The support of this optimization requires a mechanism to indicate to the cache that a barrier routine is reached to invalidate expiring copies. It also requires that barrier release contains a way to signal the directories that a barrier is released. This signal is used to reset the speculative exclusive bits. They also serve to reset sharing bits for caches having expiring copies. One message is sent to each directory. The repair process happens independently in the caches and the directories. If speculative node id is used, then the sharing list should be repaired to remove nodes having expiring copies.

This mechanism helps to avoid some of the unnecessary rollbacks. Besides the added complexity, they allow deeper speculation that may be beneficial or dangerous. Deeper speculation can be more expensive, if we eventually fail due to other types of violation. Actually, this optimization makes dependency prediction more important to avoid performance degradation.

7.4.4 Livelock Avoidance

Livelock avoidance is a by-product of some of the mechanisms described earlier. First, disallowing conflicting speculation requests and blocking speculation (Section 7.4.1) prevent speculators from ping-ponging a cache line. Also, the insertion of retry delay prevents a speculator from persistently winning the race with non-speculators, as they do not see these delays. Limiting the number of speculation attempts works as a last safeguard. This is necessary because delays that guarantee a non-speculator to

win a race can be high, due to the non-uniform memory access nature of the system. The memory line may be nearer to the speculator. Insertion of large delay can solve the problem but may make the opportunities for speculation smaller.

In summary, a statistically good value for delay can be chosen, and limiting the attempts of speculation can work as a safeguard.

7.4.5 Avoiding True Dependency Rollbacks

Slipstream mode can provide a confidence mechanism for barrier speculation, if we have a system composed of CMPs. With barrier speculation, we do not aim at having slipstream to improve the performance of barrier speculation directly. Instead, slipstream can help to detect when barrier speculation should not be performed.

The synchronization used with slipstream mode is one-token global. Actually, as the synchronization between the A-stream and the R-stream (token insertion and consumption) is executed non-speculatively, the synchronizations toggle between one-token global and zero-token global. A speculating R-stream does not insert a token until speculation is verified. In this situation, the synchronization resembles a zero-token global. Non-speculating processes run with one-token global. So, speculators have less view of the future than non-speculator.

This synchronization model serves our purpose of having the non-speculator to have a higher look-ahead. This look-ahead can be used to avoid migration of data prematurely to speculators. Those ahead (speculators) do not need this dependency information.

7.4.5.1 Avoiding Rollback Under Producer-Consumer Relation

In Chapter 6, we discussed different ways of predicting a last access or write of a cache line in a certain session. For barrier speculation, we need to predict if a line is ready for migration to a speculative process or not. If a line is predicted ready for migration, then a speculative request is satisfied. If the line is predicted to be touched in the future, then a speculative request is NACKed.

As we assume the support of expiring copies, we need to predict last write only.

Expiring copies allow us to service a non-speculative read after a speculative write. If a line is predicted last written, then upon a speculative write request we keep an expiring copy in the cache. This expiring copy can be used for subsequent reads.

Figure 7.4 shows the mechanism used to prevent premature migration from non-speculative process to a speculative process. This is based on the prediction of last write using access clustering, discussed in Section 6.5. The mechanism works as follows:

- The A-stream marks lines that will be written in the cache. If the line is not cached a future write hint is sent to the directory.
- Lines that are marked for future write are not migrated to speculative processes. The migration is denied by the holding cache or the directory. The directory future write bit is reset when the line is requested exclusively. The NACKed request indicates that the speculative requester should wait before retrying.
- When a write cluster is started by the R-stream, the future writes bit is reset. The decay counter of the cluster is re-initialized every time a new write is committed. The decay counter is decremented every fixed amount of cycles. The decay counter is a 2 bit counter in our implementation. One tick occurs every 50 cycles[2]. The A-stream should not set the future write bit for a cache line while being accessed by the R-stream (decay counter is greater than 0).
- While the line is not written, the decay counter is decremented. If a speculative request comes to the cache, it is NACKed normally. The speculator may retry after a few hundreds of cycles.

To implement this scheme, we have two bits for the decay counter. For future-write marker, we have two bits. One bit is marked by the A-stream when it is with the R-stream in the same session. The other bit is used when the A-stream is in the next session. When the R-stream enters the session that the A-stream is in, the next session future-write is copied to the current session future-write. These bits and

[2]The count of cycles per tick is better to adapt with the application behavior [22, 53].

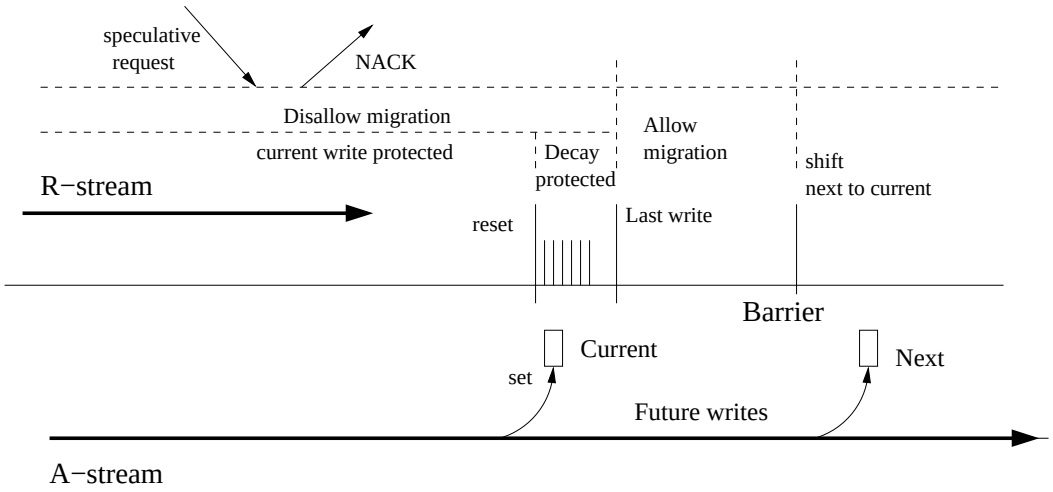


Figure 7.4: Slipstream based premature migration prevention.

counter are associated with the L2 cache. To alleviate the need for propagating every accesses of L1 cache to the L2 cache, we can use a bit to indicate that an access is sent to the L2 cache. This bit is set when an access is sent, and reset after certain number of cycles elapse (for instance, 50 cycles). In the directory, two similar bits are used to indicate future-writes.

The scheme assumes that the cache line has only one producer and possibly multiple consumers in a single session. This assumption fairly covers data that exhibit the producer-consumer sharing pattern. If the same cache line migrates, then we need to identify these lines to avoid speculation on them.

7.4.5.2 Avoiding Rollback for Migratory Data

In this section, we are interested in identifying migratory data that are written by more than one process within the same session. This class of migratory data can be detected only at the directory. Each process can know only about its own behavior and has no way to know if the line will migrate to different processes.

To identify these lines, we need at the directory a counter instead of a bit per line for future writes. A line is predicted to be migratory if the number of future write-accesses is more than 2, as shown in Figure 7.5. A 2-bit counter can serve efficiently

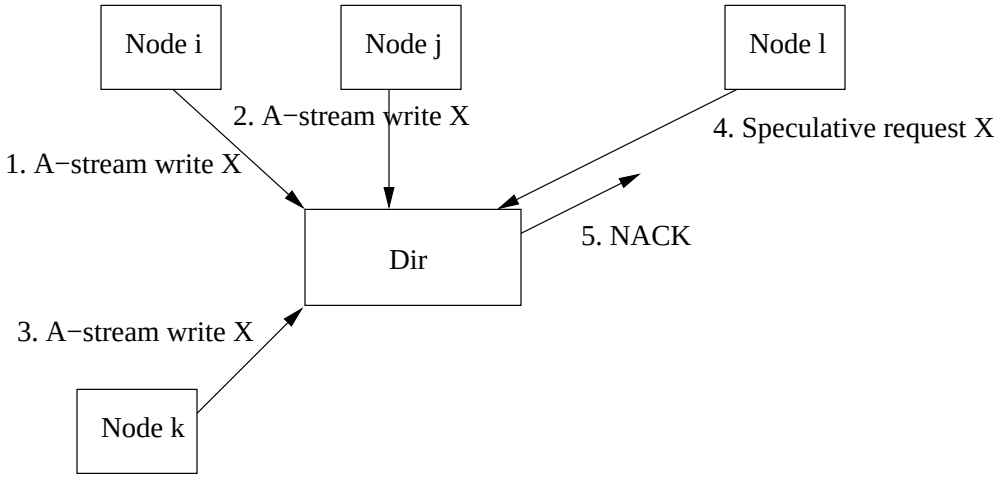


Figure 7.5: Protecting migratory data from migration to speculative process.

in this regard. The A-streams send this information to the directory, when the line is not cached. If the line is cached then it is marked in the cache. Upon replacement of a line marked by the A-stream and not accesses by the R-stream, a hint of future write accompany the message sent to the directory. Multiple future writes to the same cache line indicates that the line is migratory.

A request for speculative access for migratory shared data is denied, since we cannot predict when this line will be ready for speculation. The speculator is blocked, in this case. The speculator request will be NACKed even if the counter is less than or equal to two but this NACK does not indicate permanent blocking.

If we use a higher resolution counter that is suitable to the number of nodes, then write count incremented by the A-stream is decremented by the R-stream. When the count is zero, the cache holding the cache line can determine if the line is ready for migration. This counter will need a 4 bits for a 16-node system. In this case, a speculator is advised to retry later if the counter is larger than 0. The retry period can be proportional to the counter value. If the counter is zero, then the cache holding the line should be contacted for line readiness for migration.

Special attention should be taken when sending A-stream hints for writes. The hints sent by the same A-stream should not be repeated in one session. This reduces the network traffic and prevents misleading the directory on predicting the number

of writers. As the writes are usually very clustered, we build a table of most recently sent (MRS) cache lines. We use least recently used (LRU) to replace entries from this table. A hint for cache line is sent when it misses in the MRS table. We arbitrarily chose a table of 20 entries. The counter will be incremented if multiple processes try to access the same cache line or if there is more than one cluster of writes for the same line. While the latter does not represent migratory behavior, it presents a difficult to predict last touch scenario using the decay concept. In this case, it is good to block the speculator from accessing these cache lines.

The percentage of migratory shared data is usually very small compared to the data set. The overhead of these messages on the network can be reduced further, given that these hints are not in the critical path of execution. This allows grouping multiple hints in single transactions and possibly delaying some of them. It is also possible to append them to other transactions. They can also be treated as low priority requests by the memory network. The memory controller does not need to serialize handling these hints with other memory transactions. These hints can be handled independently in parallel, thus not increasing the occupancy of the memory controller. Occupancy is more critical to performance than bandwidth for most DSM applications [8].

Figure 7.6 summarizes the additional bits needed per cache line in the cache and the directory.

7.4.5.3 Handling A-stream Requests

In this context, the A-stream provides information about future accesses besides its prefetching duties. The A-stream can increase rollbacks if its requests force data out of cache that hold it speculatively. To solve this problem, we extend the use of transparent load, discussed in Section 5.3.1. A request by the A-stream is handled in a special way by the directory. If the line is speculatively exclusive, then an expiring copy is supplied to the requester cache (this copy is readable by the R-stream also). If expiring copies are not supported, then the action should be NACKed by the cache having the speculative copy. The directory, in this case, supplies a transparent reply,

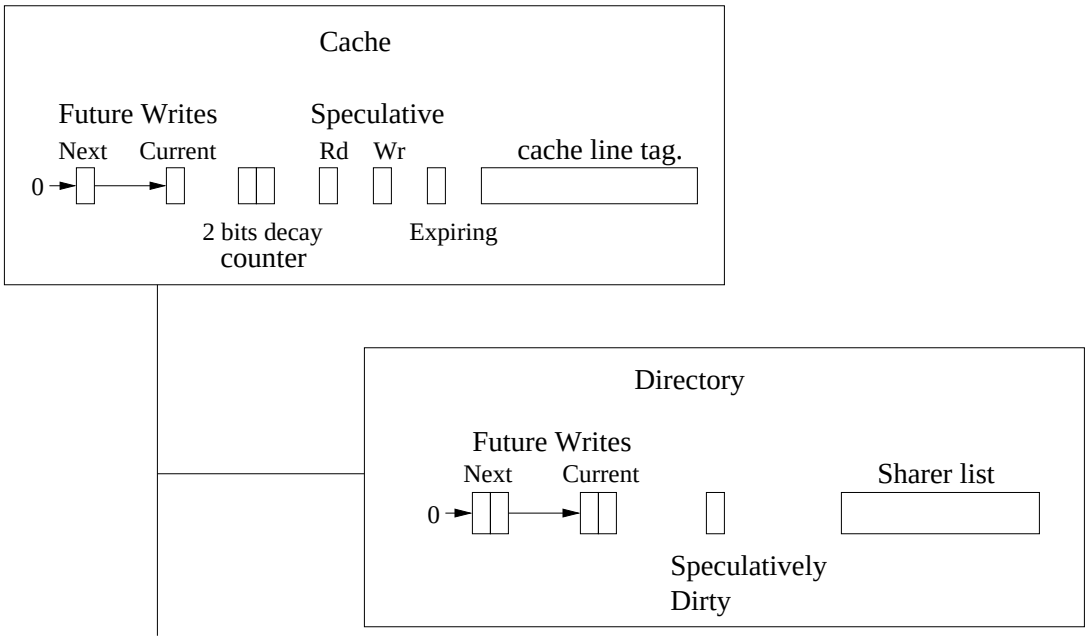


Figure 7.6: Summary of the additional bits required in the cache and the directory per cache line.

containing a speculative copy that can be accessed by the A-stream only. Additionally, the A-stream exclusive prefetch is dropped if they will lead to a rollback. We detect that a request will lead to a rollback at the directory if a speculative-exclusive bit is set or otherwise at the caches. The A-stream requests that reach the cache of a non-speculator are treated as speculative requests. These requests are NACKed if the line is not ready for migration. At the directory, these transactions are completed successfully using transparent copies (accessible by the A-stream only). The A-stream is not blocked due to data dependency because it does not do any computations.

7.5 Simulation Methodology and Results

In this section, we present the performance of four different schemes with different levels of complexity. These schemes are

- Base scheme: In this scheme, we model a speculation mechanism that runs with an unmodified coherence protocol. This scheme uses a single bit to mark any

cache line as speculative.

- + expiring copies: This scheme adds the support of expiring copies to the base scheme. Anti-dependency is broken between speculative and non-speculative processes. The coherence protocol is modified to distinguish between speculative requests and non-speculative requests. We also added a bit per line at the directory to distinguish lines held in the cache in speculatively exclusive state. The cache has a bit that differentiate between expiring and normal copies.

- +blocking: This schemes adds the blocking of speculation when the number of speculation attempts is exceeded and also upon a conflicting request to a speculative copy from another speculator.

- + dependency checking: This scheme adds dependency checking for data exhibiting a producer-consumer relation as well as migratory relation. This mode assumes that each node is a CMP. Slipstream mode is used with this scheme.

We run applications listed in Table 6.1. For applications that have more than one implementation, for example LU and OCEAN, we used the implementation with better performance.

Figure 7.7 shows the percentage of barrier time to the total execution time for 16 processes (4 processes for FFT). The barrier time ranges from 4% for FFT to 38% for SP. The barrier time is decomposed into *user barrier*, *job wait*, and *schedule wait*. We support speculation after user barriers, as they protect program data flow. Other kinds of barriers protect program control flow. Speculation after these barriers leads to the execution of wrong code, possibly harming performance. Actually, if the barrier type is of the latter kind, then we have a very limited chance for speculation. As will be shown later, equake_m has a very limited chance for speculation due to having a large schedule-wait and job-wait that interleaves with other types of barriers in a way that makes the speculation not possible most of the time.

Figure 7.8 shows the performance of different speculation schemes. LU, FFT, MG, and wupwise_m achieve a performance gain ranging from 4% to 11% without dependency checking. With slipstream mode dependency, these applications performed 9%

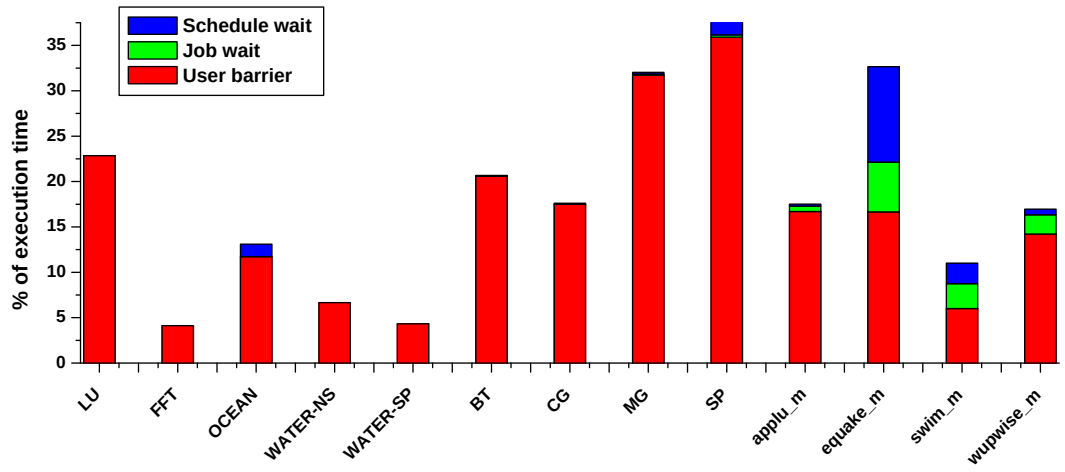


Figure 7.7: Contribution of barrier and wait time to the total execution time.

to 32% better than without speculation. The performance with slipstream dependency checking gets an additional advantage due to prefetching. LU and WATER-SP exhibit a high tolerance to frequent rollbacks. They are not memory intensive applications and that is why they show a high ability to self-repair after misspeculation. Applications like OCEAN, BT, SP, applu_m show the need for dependency checking or the performance can degrade significantly (up to -68% for BT). These applications are more memory intensive, and dependency violations can lead to frequent misspeculations. By identifying speculation attempts that lead to dependency violations, we achieved good performance gains for these applications. Dependency checking, in its full implementation, requires slipstream mode (running on a CMP). We need this mode of execution when the application is more memory intensive. This serves the purpose of prefetching as well as having a confident speculation after the barrier. It is notable also that blocking reduces the possibility of having negative performance and enhances positive performance for most applications. This is because of reducing rollbacks.

Figure 7.9 shows the percentage of successful speculation attempts and rollbacks, decomposed based on the rollback reason. These speculation attempts are normalized to the base case number of attempts. Applications like FFT, MG, swim_m, and

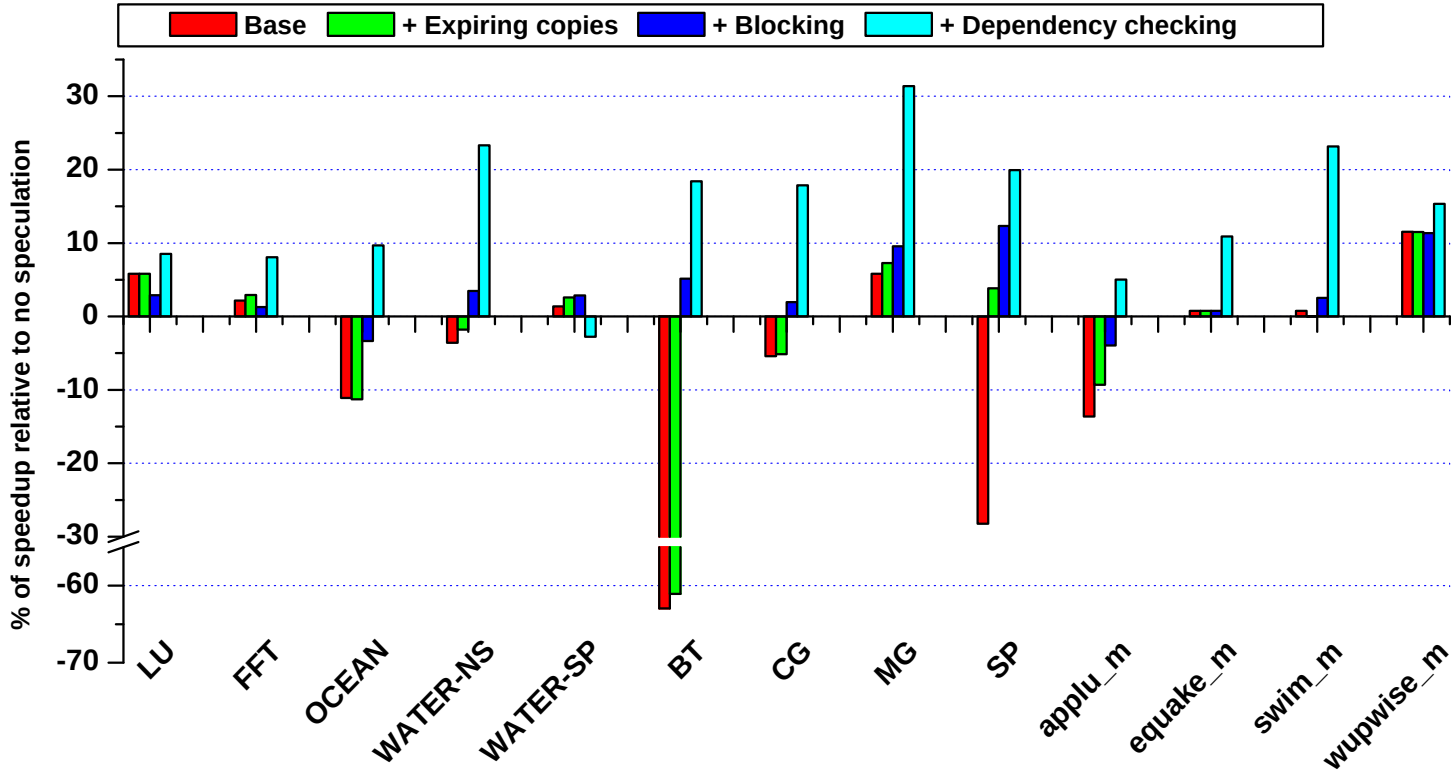


Figure 7.8: Performance of different barrier speculation mechanisms.

wupwise_m have a high percentage of successful speculation with all speculation mechanisms. Using dependency mechanisms significantly reduces the speculation attempts by 75% on average for applications like OCEAN, BT, WATER-(NS,SP), CG, SP, and applu_m. Avoiding anti-dependence rollback for applu_m using expiring copies is not very successful in reducing the total rollbacks because other dependency violations take over.

For dependency using slipstream mode, there are two sources of inaccuracy: the timing inaccuracy and the decay inaccuracy. The timing inaccuracy is due to the availability of future information after the violation. The decay inaccuracy is due to the limited coverage of inter-write time. Increasing the time required to predict no future access can reduce the rollback but can increase the blocking significantly.

Figure 7.10 shows the percentage of cache lines evicted during rollbacks to all cache lines written during speculation. The data evicted are either originally cached or brought from memory.

The average percentages of cached lines evicted due to rollback are 42% for base, 37% for +expiring, 14% for +blocking, 6% for +dependency. For memory requested lines, the percentages are 50%, 47%, 28%, and 12%, respectively. This shows that adding dependency checking reduced the amount of memory traffic unnecessarily injected into the memory system. It also preserves the useful cache state before speculation.

Figure 7.11 shows the percentage of blocking compared to the barrier time due to different blocking reasons discussed in Section 7.4.1. Cache blocking and kernel entrance blocking is supported for all speculation configurations. Another speculator blocking and maximum rollback blocking is supported for +blocking scheme. Migratory blocking is active only with +dependency. This figure does not show blocking due to caches, as they are retried by the speculator after a certain period of time and do not get blocked to the end of the session. The blocking is a percentage of the barrier time in each individual run (not normalized).

It is noticeable that 84% of the blocking is due to kernel entrance. This kernel entrance happens when a speculator enters another synchronization routine, I/O routine, scheduling routine, etc. This kind of blocking cannot be dealt with at the

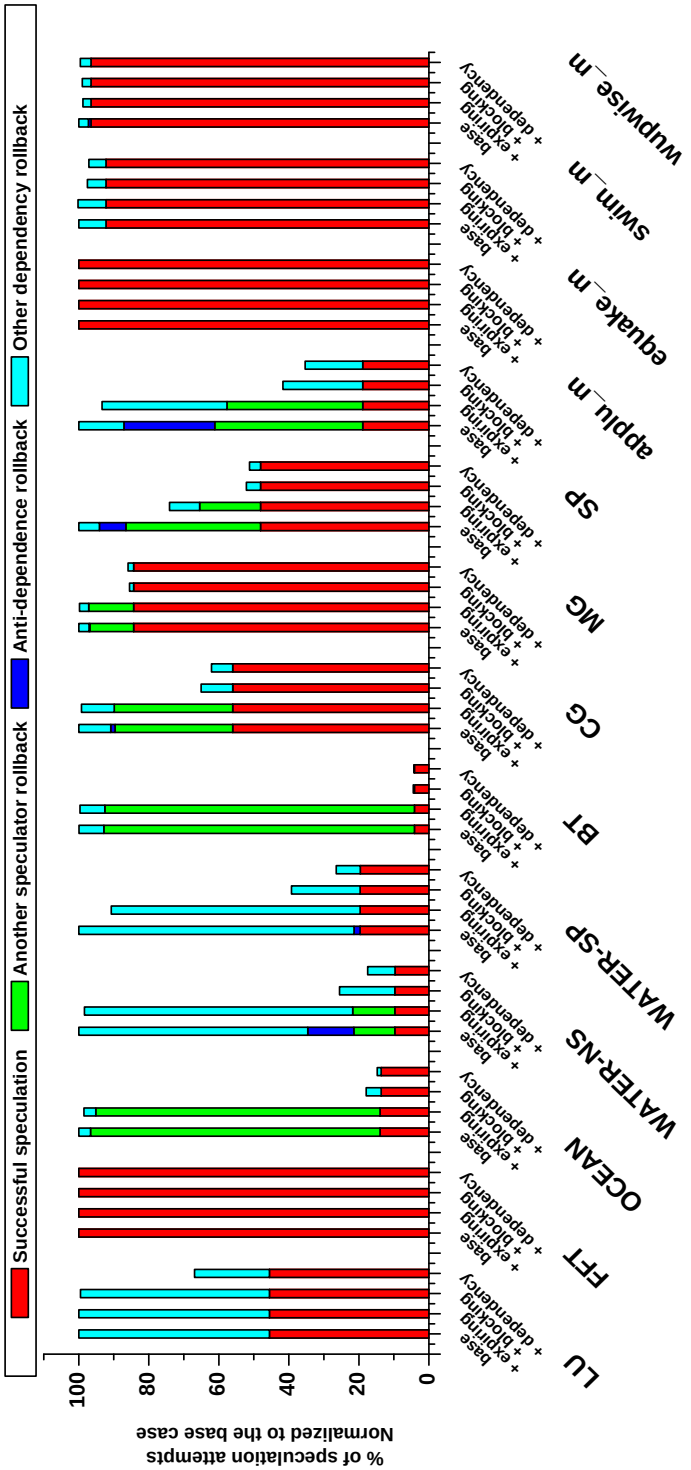


Figure 7.9: Speculations attempts and rollback decomposition.

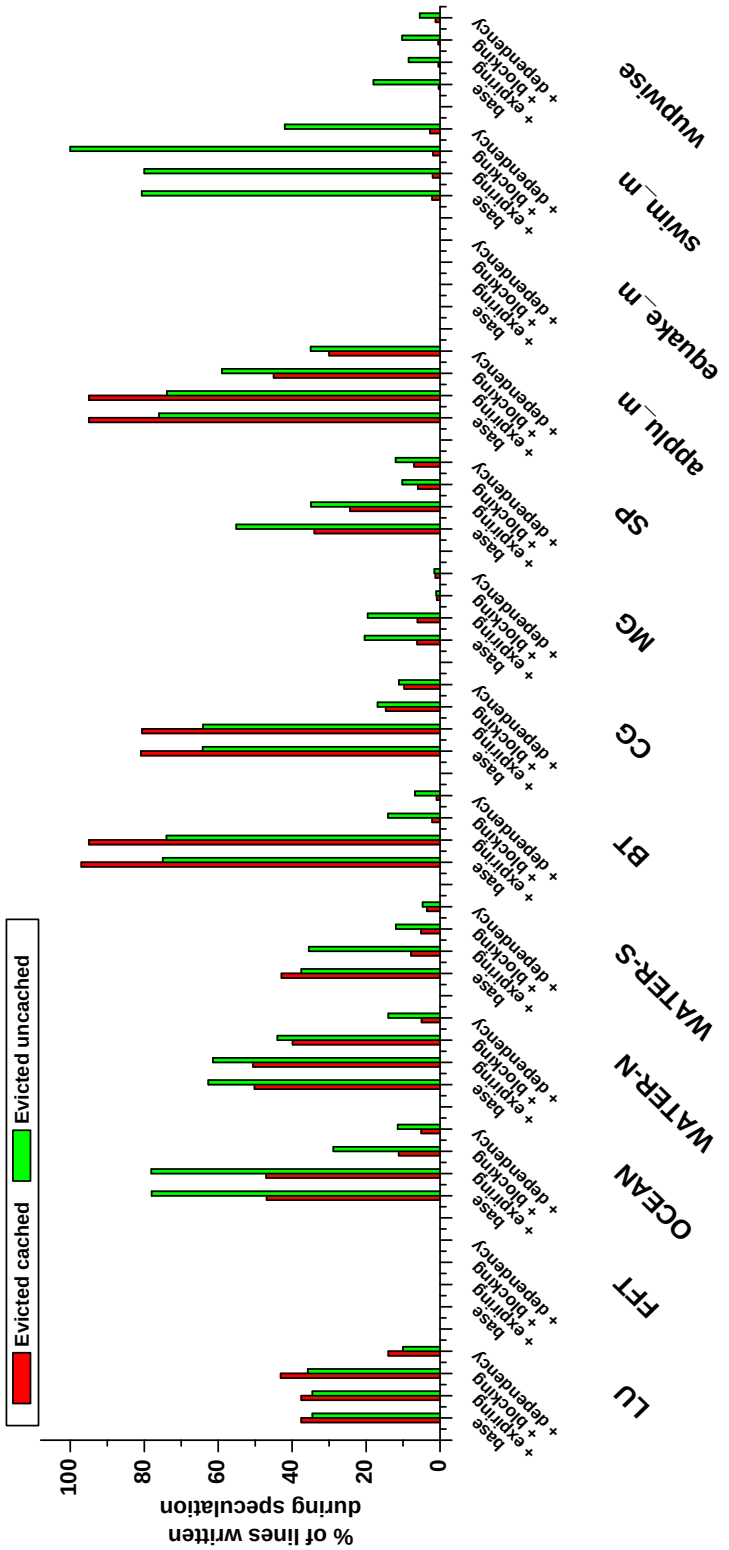


Figure 7.10: Rollack evicted lines for cached and uncached data.

hardware level. The cache blocking is due to exhausting the maximum allowed cache lines for speculation per set. This does not show up much for most benchmarks due to the large cache used for simulation compared to the problem sizes. Other dependency related blocking reduces the rollback possibility as discussed earlier.

Figure 7.12 shows the decomposition of data accessed during speculation. The figure shows the decomposition both for data read speculatively and data written speculatively. It is noticeable that the data requested from the memory gets smaller with more confident speculations. In fact, for data read the percentage of data requested from memory dropped from 28% on average for the base to 12% on average for +dependency. For data written during speculation the average dropped from 48% for base to 28% for +dependency. The data read while being dirty in the cache is about 40% for confident speculation. In our implementation, we do not require updating the memory for these cache lines as we use two bits to differentiate between speculative reads and speculative writes of the cache line. For data speculatively written, we notice that there is a high percentage of data read before being written. Differentiating between reads and writes allows us to delay updating these cache lines to the memory. This can be advantageous if a rollback happens after the read accesses and before the write accesses.

Figure 7.13 compares the performance of using barrier speculation with dependency checking based on slipstream mode and using the CMP to run 2 tasks. The figure also shows the performance of running slipstream mode with the same synchronization (one-token global) without barrier speculation. This approximates the effect of prefetching to some extent. This does not isolate the effect of prefetching, as speculating processes run in zero-token global synchronization while non-speculating processes run in one-token global. This behavior is due to the insertion of token only non-speculatively. Additionally, the A-stream requests are handled in a special way to avoid rollbacks, as discussed in Section 7.4.5.3.

It is notable that applications that are less memory intensive (like LU and WATER-SP) are the ones that do not benefit from dependency checking during speculation. Applications that are more memory dependent benefit from slipstream both for prefetching and for dependency prediction for speculation.

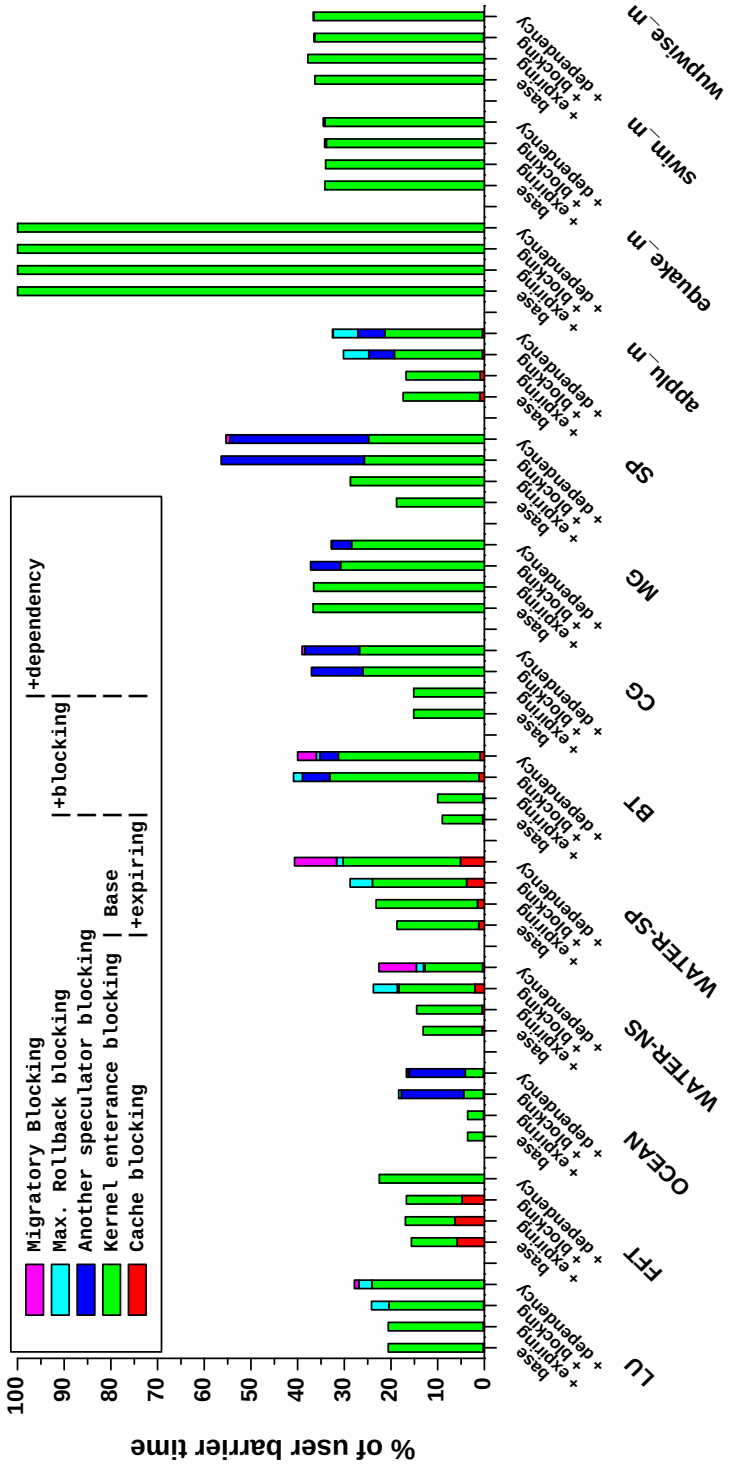


Figure 7.11: Causes for speculation blocking for different schemes.

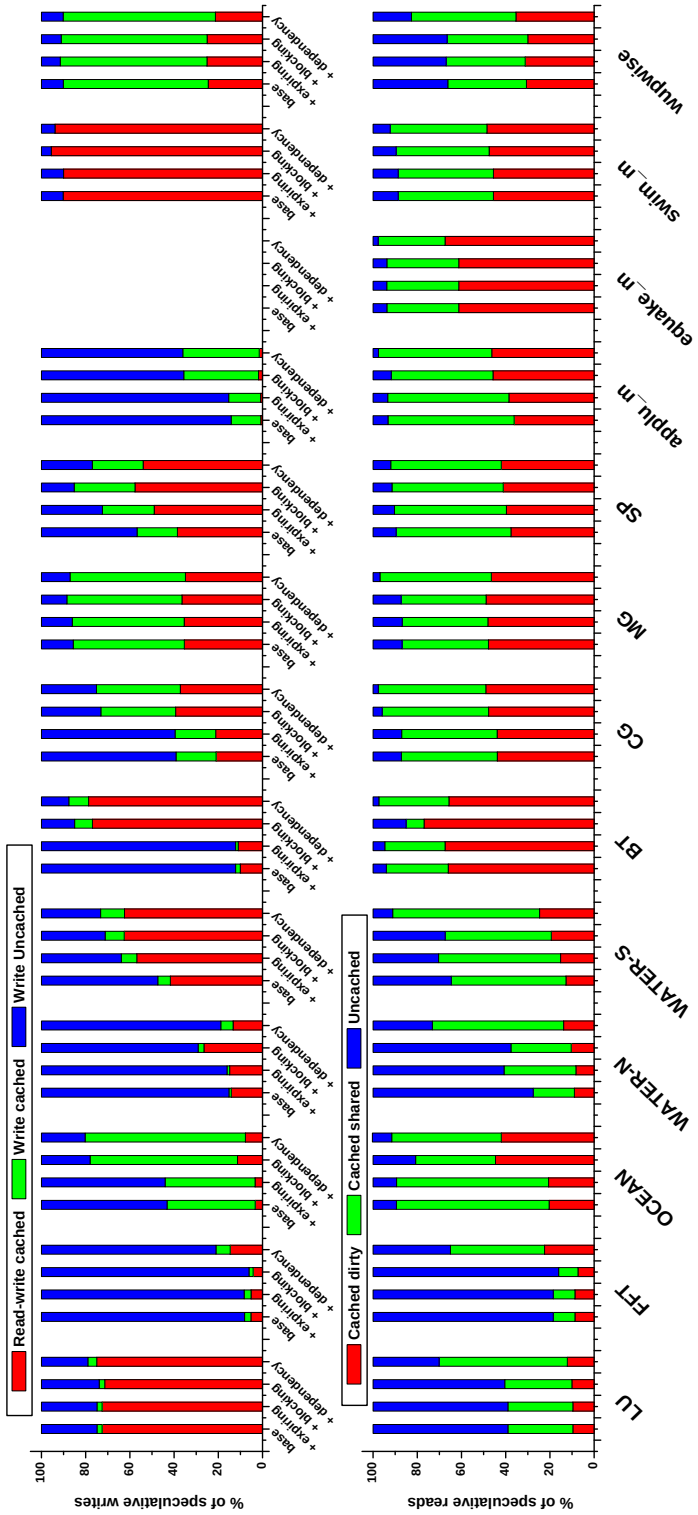


Figure 7.12: Decomposition of data read and written during speculation for different schemes.

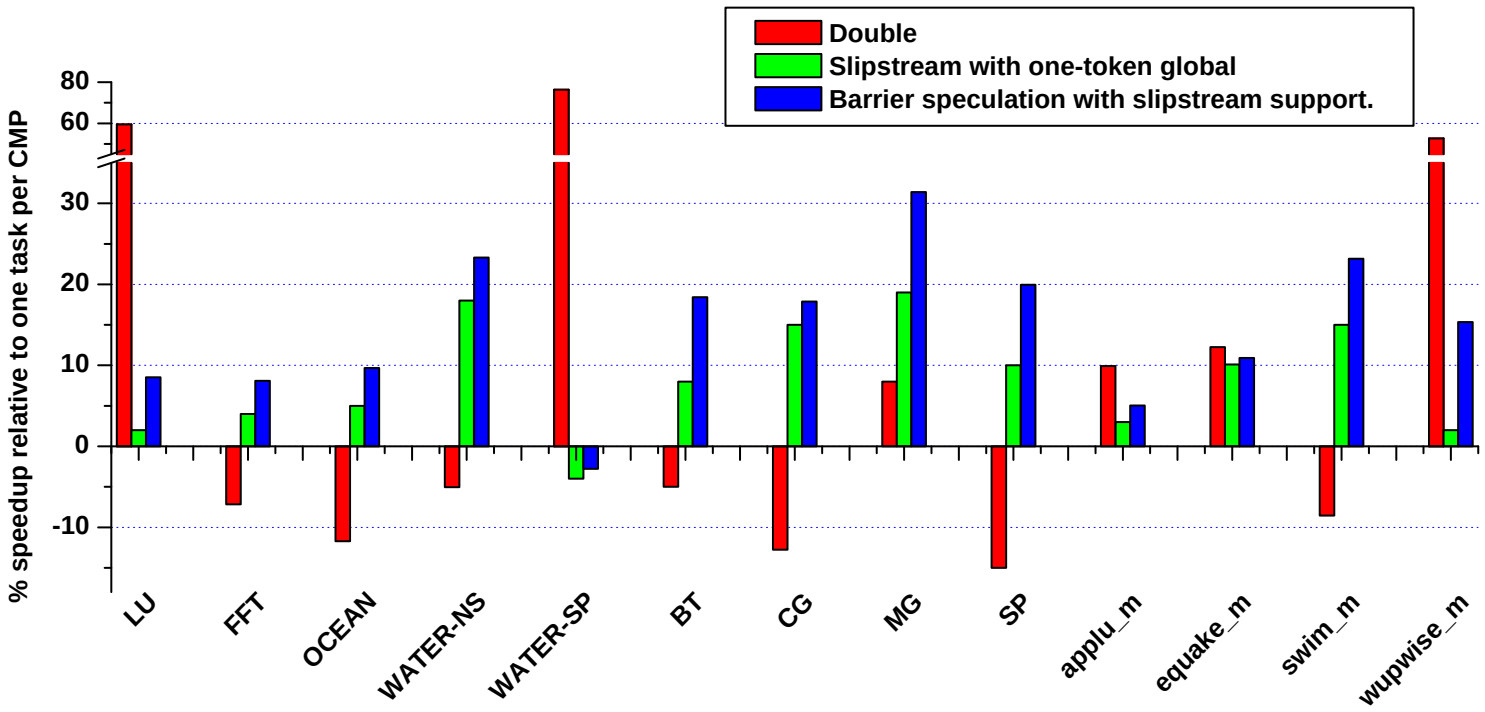


Figure 7.13: Performance of barrier speculation with dependency checking *vs.* using the CMP differently.

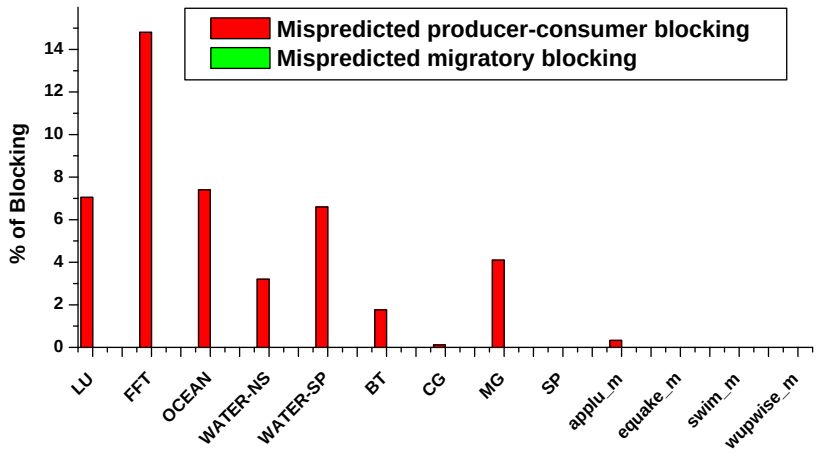


Figure 7.14: False blocking due to dependency checking.

Figure 7.14 shows the percentage of false positive blocking. A cache line is falsely blocked if a speculator request is NACKed and the line is not rewritten by a non-speculator until hitting the barrier routine. If the blocking is due to the cache then we designate this as *producer-consumer blocking*. We also call blocking by the directory *migratory blocking*.

As shown in Figure 7.14, the false producer-consumer blocking is small (<15% for all applications with an average of 3.5%). For Migratory blocking, the false blocking is negligible. It is worth mentioning that producer-consumer blocking is dependent on the accuracy of A-stream and the decay of accesses concept. For migratory blocking, it is solely dependent on A-stream accuracy.

Chapter 8

Conclusions and Future Work

Slipstream execution mode in a CMP-based multiprocessor enables the construction of a program-based view of the future to attack coherence, communication, and synchronization overheads. Slipstream mode uses the additional processing power of a CMP node to more efficiently communicate among parallel tasks, rather than to increase task-level concurrency. In this work, we have introduced a method for creating a shortened A-stream by skipping synchronization and not committing shared memory stores. We also describe mechanisms for locally synchronizing the A-stream and R-stream as needed, and for recovering a deviating A-stream. This basic model allows the A-stream to run ahead and generate prefetches to shared data that benefit the R-stream. Slipstream-based prefetching performs up to 19% better than running either one or two tasks per CMP on systems with 16 CMPs.

We also introduce the concept of a transparent load, which allows the A-stream to make correct forward progress while minimizing premature migration of exclusively owned cache lines. Transparent loads are also used as hints of future sharing behavior, and we describe a form of self-invalidation that exploits these hints. When transparent loads and SI are added to prefetching, slipstream mode is up to 29% faster than the best of running one or two tasks per CMP.

This work explores the opportunity for transparent support of slipstream execution mode using OpenMP. This minor extension of OpenMP allows applying slipstream transparently on wide range of parallel applications. We introduce how to extend a

compiler to support this mode. We also discuss how to handle each OpenMP directive, taking special care of the semantics of each directive, to achieve good performance. The extension requires modification of the internal threading library and mostly does not affect the code transformation phase and other optimizations performed by the compiler. A simple directive is needed to control slipstream behavior.

Our implementation allows the same binary to run in different modes, based on the application's need. Slipstream mode is an additional mode that proves to be useful when communication overhead dominates the execution time.

We also investigated the interaction between slipstream mode and dynamic scheduling. This requires more restrictive synchronization between the streams, since the A-stream cannot run ahead until it knows the work that is assigned to its R-stream. Nevertheless, dynamic scheduling often results in increased communication and data migration overheads, both of which can potentially be reduced by slipstream execution mode. Our study shows an average performance gain of 12% for the four dynamically-scheduled benchmarks.

OpenMP, with its relative ease of use, opens the door to have more shared memory applications. Slipstream execution mode can extend the scalability of those applications on CMP-based multiprocessors, by applying additional processors to reduce communication overheads. This research demonstrates that a combination of OpenMP and slipstream mode can benefit both programmers and end users, improving the performance of portable parallel applications.

Addressing the overhead due to synchronization, slipstream mode provides a mechanism to avoid frequent rollbacks during barrier speculation. Slipstream mode can help to identify dependencies. Identifying dependencies helps blocking/delaying transactions that can lead to rollbacks. This can reduce rollbacks by up to 95%. Avoiding rollbacks reduces the overheads due to wasted memory requests, which can harm performance. It also preserves cached data before/during speculation. Adding a confidence mechanism to barrier speculation helps to improve speedups by up to 13%.

8.1 Pros and Cons of Slipstream Execution Mode

The following attributes constitute what is unique about this research proposal:

- The coupling between the predictive thread (A-stream) and the computing thread (R-stream) is loose enough to generate far-ahead future information. The complexity of producing these future information is independent of the latency. Tightly-coupled future prediction schemes are expected to not scale as the latency increases. This scheme promises a scalable future-based prefetching. We mean by scalability that the scheme can adapt as the number of cycles required for memory access increases because it uses memory latency to be ahead.

- Simple formation of the A-stream. The proposal combines formation of the stream using hardware support and the ability to be far ahead. The simplicity is due to observing the roles of shared variables *vs.* local variables in parallel programs.

- The proposal achieves very high functional accuracy almost 100% in most cases. We mean by functional accuracy the matching between the memory accesses provided by the A-stream and those generated by the R-stream.

On the other hand, the following challenges faces this proposal:

- Coupling the R-stream and A-stream to achieve timing accuracy. We mean by the timing accuracy providing the information in timely manner. For example, prefetching should not be too far ahead or too late.

- Limited applicability. Slipstream mode is beneficial when parallel programs reach their scalability limit. Programs with embedded scheduling code and user synchronization cannot be ported transparently. Programmer intervention may be needed to port these applications to use slipstream mode. Compiler-assisted parallelization, such as OpenMP, does not face this problem.

8.2 Future Work

One of our future goals is to create development and run-time environments that allow users to choose the best mode to efficiently utilize system resources. We are also interested in extending the analysis to recommend an A-R synchronization scheme for a given program, or even varying the scheme dynamically during program execution. We will consider developing a different timing model to synchronize between the A-stream and R-stream. We intend to extend the A-stream ability to run with multiple speeds without the need to violate dependencies imposed by synchronizations.

We think also that this concept can applied to other programming model, for example software DSM. Software DSM usually suffers from long latencies that reduces the efficiency of computational resources.

Bibliography

[1] Omni OpenMP Compiler Project. http://phase.etl.go.jp/Omni/.

[2] OpenMP specifications. http://www.openmp.org/specs/.

[3] SPEC OMP 2001 Benchmark Suit. http://www.spec.org/omp.

[4] Adiga, N. R., et al. An Overview of The BlueGene/L Supercomputer. *Proceedings of the Supercomputing Conference* (Feb. 2002).

[5] Annavaram, M., Patel, J., and Davidson, E. Data Prefetching by Dependence Graph Precomputation. *28th Int'l Symp. on Computer Architecture* (July 2001).

[6] Balasubramonian, R., Dwarkadas, S., and Albonesi, D. H. Dynamically Allocating Processor Resources Between Nearby and Distant ILP. *28th Int'l Symp. on Computer Architecture* (July 2001).

[7] Bircsak, J., Craig, P., Crowell, R., Cvetanovic, Z., Harris, J., Nelson, C. A., and Offner, C. D. Extending OpenMP for NUMA Machines. *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing* (2000), 48.

[8] Chaudhuri, M., Heinrich, M., Holt, C., Singh, J. P., Rothberg, E., and Hennessy, J. Latency, Occupancy, and Bandwidth in DSM Multiprocessors: A Performance Evaluation. *Computer Systems Laboratory Technical Report CSL-TR-2002-1025, Cornell University* (July 2002).

[9] Chen, D. K., and Yew, P. C. Redundant Synchronization Elimination for Doacross Loops. *IEEE Trans. on Parallel and Distributed Systems 10*, 5 (May 1999).

[10] Chen, T.-F., and Baer, J.-L. Effective Hardware-Based Data Prefetching for High-Performance Processors. *IEEE Transactions on Computers 44*, 5 (1995).

[11] Collins, J., Tullsen, D., and Wang, H. Dynamic Speculative Precomputation. *34th Int'l Symp. on Microarchitecture* (Dec. 2001).

[12] Collins, J., Wang, H., and Tullsen, D. Speculative Precomputation: Long-range Prefetching of Delinquent Loads. *28th Int'l Symp. on Computer Architecture* (July 2001), 14–25.

[13] Dundas, J., and Mudge, T. Improving Data Cache Performance by Pre-executing Instructions Under a Cache Miss. *Int'l Conference on Supercomputing* (July 1997).

[14] Eichenberger, A. E., and Abraham, S. G. Modeling Load Imbalance and Fuzzy Barriers for Scalable Shared-memory Multiprocessors. *Hawaii Intl. Conf. on System Sciences I* (June 1995), 262–271.

[15] Fisher, J. VLIW Architectures: An Inevitable Standard for the Future? *Journal of Supercomputer 7*, 2 (Mar. 1990), 29–36.

[16] Herrod, S., et al. *The SimOS Simulation Environment.* http://simos.-stanford.edu/userguide/, Feb. 1998.

[17] Ibrahim, K. Z., and Byrd, G. T. On the Exploitation of Value Prediction and Producer Identification to Reduce Barrier Synchronization Time. *Int'l Parallel and Distributed Processing Symp.* (Apr. 2001).

[18] Ibrahim, K. Z., and Byrd, G. T. Extending OpenMP to Support Slipstream Execution Mode. *Proceeding of the 17th International Parallel and Distributed Processing Symposium* (Apr. 2003).

[19] Ibrahim, K. Z., Byrd, G. T., and Rotenberg, E. Slipstream Execution Mode for CMP-Based Multiprocessors. *9th Int'l Conf. on High-Performance Computer Architecture* (Feb. 2003).

[20] Kahle, J. Power4: A Dual-CPU Processor Chip. *Microprocessor Forum* (Oct. 1999).

[21] Kaxiras, S., and Goodman, J. R. Improving CC-NUMA Performance Using Instruction-Based Prediction. *5th Int'l Symp. On High-Performance Computer Architecture* (1999), 161–170.

[22] Kaxiras, S., Hu, Z., and Martonosi, M. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. *28th Int'l Symp. on Computer Architecture* (July 2001), 240–251.

[23] Kusano, K., Satoh, S., and Sato, M. Performance Evaluation of the OMNI OpenMP Compiler. *Lecture Notes in Computer Science 1940* (2000), 403.

[24] Lai, A.-C., and Falsafi, B. Memory Sharing Predictor: The Key to a Speculative Coherent DSM. *Proc. of the 26th Annual Int'l Symp. on Computer Architecture* (July 1999).

[25] Lai, A.-C., and Falsafi, B. Selective, Accurate, and Timely Self-Invalidation Using Last-Touch Prediction. *27th Int'l Symp. on Computer Architecture* (June 2000), 139–148.

[26] Laudon, J., and Lenoski, D. The SGI Origin: A ccNUMA Highly Scalable Server. *24th Int'l Symp. on Computer Architecture* (June 1997), 241–251.

[27] Lebeck, A. R., and Wood, D. A. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared Memory Multiprocessors. *22nd Int'l Symp. on Computer Architecture* (June 1995), 48–59.

[28] Lenoski, D., Laudon, J., Gharachorloo, K., Gupta, A., and Hennessy, J. L. The Directory-based Cache Coherence Protocol for the DASH

Multiprocessor. *17th Int'l Symp. on Computer Architecture* (May 1990), 148–159.

[29] Luk, C.-K. Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors. *28th Int'l Symp. on Computer Architecture* (July 2001), 40–51.

[30] Luk, C.-K., and Mowry, T. C. Compiler-Based Prefetching for Recursive Data Structures. *7th Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems* (Oct. 1996), 222–233.

[31] Lusk, E., and et al. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, New York, 1987.

[32] Martinez, J. F., and Torrellas, J. Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications. *10th Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems* (2002).

[33] Merlin, J. Distributed OpenMP: Extensions to OpenMP for SMP Clusters. *Second European Workshop on OpenMP* (2000).

[34] Michael, M. M. High Performance Dynamic Lock-free Hash Tables and List-based Sets. *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures* (2002), 73–82.

[35] Mowry, T. C., and Gupta, A. Tolerating Latency through Software-Controlled Prefetching in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Processing 12*, 2 (June 1991), 87–106.

[36] Mukherjee, S. S., and Hill, M. D. Using Prediction to Accelerate Coherence Protocols. *25th Int'l Symp. on Computer Architecture* (June 1998).

[37] Nikolopoulos, D., Artiaga, E., Ayguadé, E., and Labarta, J. Exploiting Memory Affinity in OpenMP through Schedule Reuse. *3rd European Workshop on OpenMP* (Sept. 2001).

[38] Olukotun, K., Nayfeh, B., Hammond, L., Wilson, K., and Chang, K.-Y. The Case for a Single-Chip Multiprocessor. *7th Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems* (Oct. 1996).

[39] Philippsen, M., and Heinz, E. A. Automatic Synchronization Elimination in Synchronous FORALLs. *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation* (Feb. 1995).

[40] Rajwar, R., and Goodman, J. R. Transactional Lock-Free Execution of Lock-Based Programs. *10th Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems* (2002).

[41] Rosenblum, M., Bugnion, E., Devine, S., and Herrod, S. A. Using the SimOS Machine Simulator to Study Complex Computer Systems. *Modeling and Computer Simulation 7*, 1 (1997), 78–103.

[42] Roth, A., Moshovos, A., and Sohi, G. S. Dependence-Based Prefetching for Linked Data Structures. *8th Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems* (Oct. 1998).

[43] Roth, A., and Sohi, G. S. Speculative Data-Driven Multithreading. *7th Int'l Conf. on High-Performance Computer Architecture* (Jan. 2001), 191–202.

[44] Sato, T., Ohno, K., and Nakashima, H. A Mechanism for Speculative Memory Access following Synchronization Operations. *14th Int'l Parallel and Distributed Processing Symp* (2000), 145–154.

[45] Silicon Graphics, Inc. *SGI 3000 Family Reference Guide*. http://www.sgi.-com/origin/3000/3000_ref.pdf, 2000.

[46] Smith, J. Decoupled Access/Execute Computer Architecture. *9th Int'l Symp. on Computer Architecture* (July 1982).

[47] Solihin, Y., Lee, J., and Torrellas, J. Prefetching in an Intelligent Memory Architecture using Helper Threads. *5th Workshop on Multithreaded Execution, Architecture, and Compilation* (Dec. 2001).

[48] Sundaramoorthy, K., Purser, Z., and Rotenberg, E. Slipstream processors: Improving both Performance and Fault Tolerance. In *Architectural Support for Programming Languages and Operating Systems* (2000), pp. 257–268.

[49] Tomasulo, R. M. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development* (Jan. 1967), 25–33.

[50] Tullsen, D. M., Eggers, S. J., and Levy, H. M. Simultaneous Multithreading: Maximizing on-Chip Parallelism. *Proceedings of the 22nd Annual International Symposium on Computer Architecture* (1995), 392–403.

[51] Valois, J. D. Lock-Free Linked Lists Using Compare-and-swap. In *Symposium on Principles of Distributed Computing* (1995), pp. 214–222.

[52] Woo, S., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A. The SPLASH-2 Programs: Characterization and Methodological Considerations. *22nd Int'l Symp. on Computer Architecture* (June 1995), 24–36.

[53] Zhou, H., Toburen, M. C., Rotenberg, E., and Conte, T. M. Adaptive Mode Control: A Static-Power-Efficient Cache Design. *International Conference on Parallel Architectures and Compilation Techniques* (Sept. 2001).

[54] Zilles, C., and Sohi, G. Execution-based Prediction Using Speculative Slices. *28th Int'l Symp. on Computer Architecture* (July 2001).