Slipstream Execution Mode for CMP-Based Multiprocessors

Khaled Z. Ibrahim, Gregory T. Byrd, and Eric Rotenberg
Dept. of Electrical and Computer Engineering, North Carolina State University
{kzmousta, gbyrd, ericro}@ece.ncsu.edu

Abstract

Scalability of applications on distributed shared-memory (DSM) multiprocessors is limited by communication 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 with shared L2 cache. A 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, running 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 two 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, e.g., self-invalidation. For multiprocessor systems with up to 16 CMP nodes, slipstream mode outperforms running one or two conventional tasks per CMP in 7 out of 9 parallel scientific benchmarks. Slipstream mode is 12-19% faster with prefetching only and up to 29% faster with self-invalidation enabled.

1. Introduction

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 paper, we consider the use of a dual-processor chip multiprocessor (CMP) [9,18] as the building block of a distributed shared memory multiprocessor. A conventional way to use such a machine is to assign a parallel task to each processor. As we approach the performance threshold, however, increasing concurrency does not help. Figure 1 shows the relative performance of assigning two tasks per CMP, compared to assigning only a single task, leaving one processor idle. 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. In such a situation, we propose 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, which uses redundant execution to speed up sequential programs [26]. A slipstream

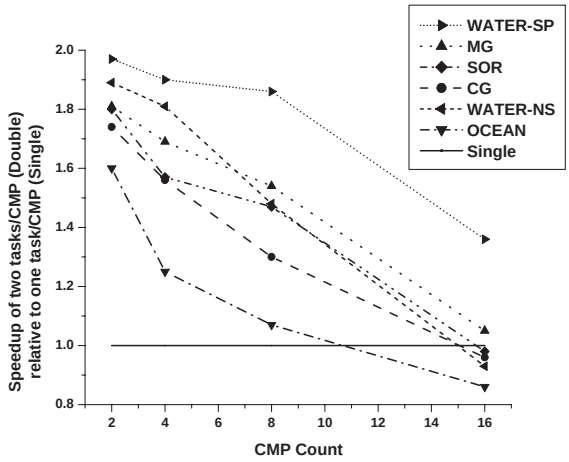


Figure 1. Speedup of two tasks per CMP (double mode) *vs*. one task per CMP (single mode).

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. For example, the A-stream provides 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.

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. This simple use of redundant computation requires very little hardware support; it is a new *mode of execution* for multiprocessor systems, rather than a new architecture.

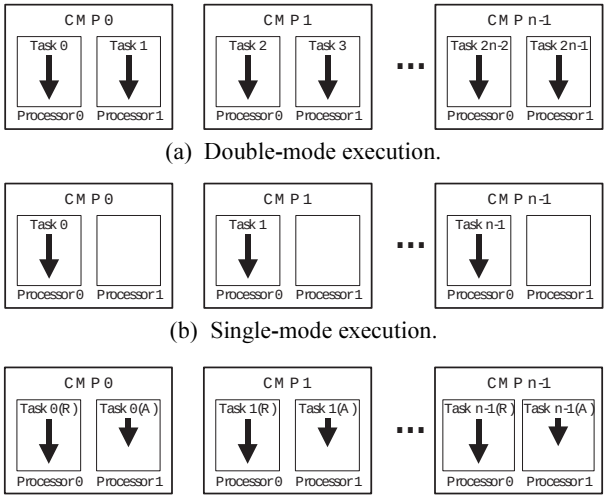
Figure 2 illustrates three different modes of concurrent execution for a system with *n* CMPs: double, single, and slipstream.

(a) 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.

(b) In *single mode*, only one task is assigned to each CMP. One processor runs the task, while the other processor is idle. As shown in Figure 1, 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.

(c) 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). The A-stream gets ahead of its R-stream by skipping synchronization events and stores to shared memory. 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.

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



(a) Double-mode execution.

(b) Single-mode execution.

(c) Slipstream-mode execution.

Figure 2. Execution modes for CMP-based multiprocessors.

can be used for coherence optimizations, such as *self-invalidation (SI)* [12]. 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. In seven out of nine benchmarks, slipstream mode is faster than both single and double mode on systems with up to 16 CMPs. Slipstream mode performs 12-29% better than the next best mode (single or double). Because it is simply a mode of execution, slipstream can be enabled or disabled, according to the needs of particular applications or critical application kernels. It offers a new opportunity for programmer-directed optimization.

Prefetching and SI are only the beginning, however. Slipstream mode enables a number of additional optimizations that benefit from accurate, dynamic prediction of memory accesses. Examples include barrier speculation [22] and migratory sharing optimizations [10]. Program-based prediction is potentially more accurate than history-based predictors and prefetchers, does not require custom auxiliary hardware tables, and does not require detailed programmer or compiler analysis. Slipstream-mode execution is a general program-based prediction mechanism that can enable

Table 1: SimOS machine parameters.

CPU: MIPSY-based CMP model, 1 GHz	
L1 Caches (I/D): 32 KB, 2-way assoc., 1-cycle hit	**L2 Cache (unified):** 1 MB**, 4-way assoc., 10-cycle hit
Memory (cycles)	Description
BusTime: 30	transit, L2 to directory controller (DC)
PILocalDCTime: 60	occupancy of DC on local miss
PIRemoteDCTime: 10	occupancy of local DC on outgoing miss
NIRemoteDCTime: 10	occupancy of local DC on incoming miss
NILocalDCTime: 60	occupancy of remote DC on remote miss
NetTime: 50	transit, interconnection network
MemTime: 50	latency, for DC to local memory

** A 128-KB cache is used for Water to match its small working set.

multiple optimization strategies to be applied selectively and simultaneously, extending the scalability of programs with minimum programmer involvement.

In Section 2, we describe the simulation framework and methodology used for the performance studies in later sections. Section 3 describes the basic slipstream mode, including the mechanisms for shortening the A-stream, correcting A-stream deviations, and managing local synchronization between the A-stream and R-stream. This basic mode allows prefetching of shared data based on program execution. Section 4 extends slipstream mode to include transparent loads and self-invalidations. Related work is discussed in Section 5, and Section 6 outlines areas for future investigation.

2. Base Architecture and Simulation Methodology

To explore the performance of slipstream execution mode, we simulate a CMP-based multiprocessor. Each processing node consists of a CMP and a portion of the globally-shared memory. The 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 [7,19], with IRIX 5.3 and a MIPSY-based CMP model. Table 1 shows the memory and network latency parameters, which are chosen to approximate the Origin 3000 memory system [23]. The minimum latency to bring data into the L2 cache on a remote miss is 290 cycles, assuming no contention. A local miss requires 170 cycles. We modified the SimOS L2 cache module to support CMP nodes. The shared L2 cache manages coherence between its L1 caches and also merges their requests when appropriate.

Nine benchmarks and their data sizes are shown in Table 2, representing kernels of scientific and engineering applications. CG, MG, and SP are shared memory versions of benchmarks from the NAS Parallel Benchmarks [29]. Except for SOR, the others are taken from Splash-2 [27].

Table 2: Benchmarks and data set sizes.

Application	**Size**
FFT	64K complex double
Ocean	258×258
Water-NS (n-squared)	512 molecules
Water-SP (spatial)	512 molecules
SOR	1024×1024
LU	512×512
CG	1400
MG	32×32×32
SP	16×16×16

3. Using Slipstream Mode for Prefetching

In this section, we describe the most basic use of slipstream mode, in which the A-stream and the R-stream communicate only through their shared L2 cache and through a simple local synchronization mechanism. The A-stream acts as a prefetch engine for the R-stream, pre-loading shared data into the L2 cache. Using slipstream mode for prefetching requires no changes to the memory subsystem or to the coherence protocol.

Section 3.1 describes basic slipstream operation. The A-stream is forked as a copy of the R-stream. By skipping synchronization events and stores to shared data, the A-stream runs ahead of the R-stream and prefetches shared data. The reduction of the A-stream is speculative and may lead to erroneous computations. We present evidence, however, that errors in local computation have minimal effect on the control flow and shared memory address calculations of the A-stream, so its predictions remain accurate.

Section 3.2 describes a loose, local synchronization mechanism, called *A-R synchronization*, that constrains how far an A-stream may run ahead of its corresponding R-stream. The R-stream also checks for significant deviations in the A-stream's control flow at A-R synchronization points and restarts the A-stream, if necessary. Section 3.3 discusses the interaction between A-R synchronization and prefetching. Running far ahead hides longer latencies, but it also increases the chance of premature migration of data. Section 3.4 discusses the performance of slipstream prefetching.

3.1 Basic slipstream operation

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.

Each A-stream task must be shortened. 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 [15] 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 prevent the propagation of speculative values produced by the A-stream by discarding stores to shared memory. (The store instruction is executed in the processor pipeline, but it is not committed.) 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 shortens the A-stream further, since shared memory stores might otherwise incur long latencies due to invalidation requests and acknowledgements.

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 through the normal coherence mechanism, and the R-stream will retrieve the correct value when needed.

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. A unique task ID identifies 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.

If shared variables do significantly impact control flow, then the A-stream's execution may diverge from the R-stream's. If they impact address generation, then the cache lines prefetched by the A-stream may be of no use to the R-stream. In either case, the A-stream is not beneficial to the R-stream, because its program-based predictions are inaccurate. 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*, 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 does not cause a divergence of the A-stream and R-stream.
- *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.

3.2 A-R synchronization and recovery

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

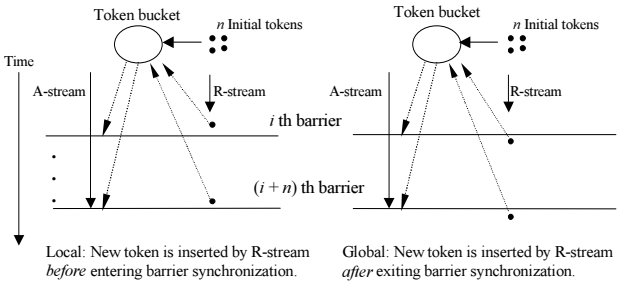


Figure 3. A-R synchronization.

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 often 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. 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. This can be viewed as creating an initial pool of tokens that are consumed as the A-stream enters a new session (Figure 3). 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.

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. This is a software-only check, and it 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.

Finally, 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.

3.3 Slipstream-based 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.

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,

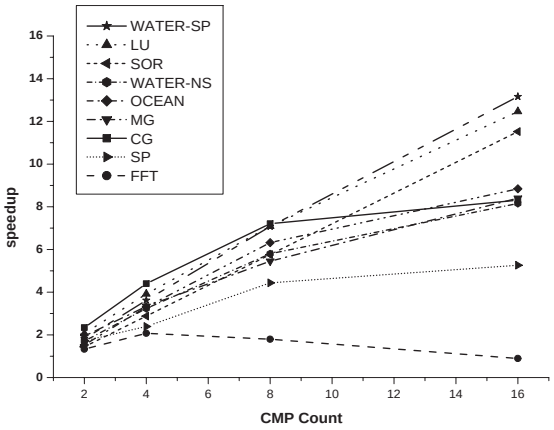


Figure 4. Speedup of single mode.

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.

The A-stream task converts some skipped stores into exclusive prefetches, if it is in the same session as the R-stream and 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 more likely to conflict with R-stream accesses from the earlier session (or critical section). For this reason, 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.

3.4 Performance of slipstream-based prefetching

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 4 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 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 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 exits the previous synchronization event.

Consider the first group of benchmarks (LU, Water-SP, and SOR), which show 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

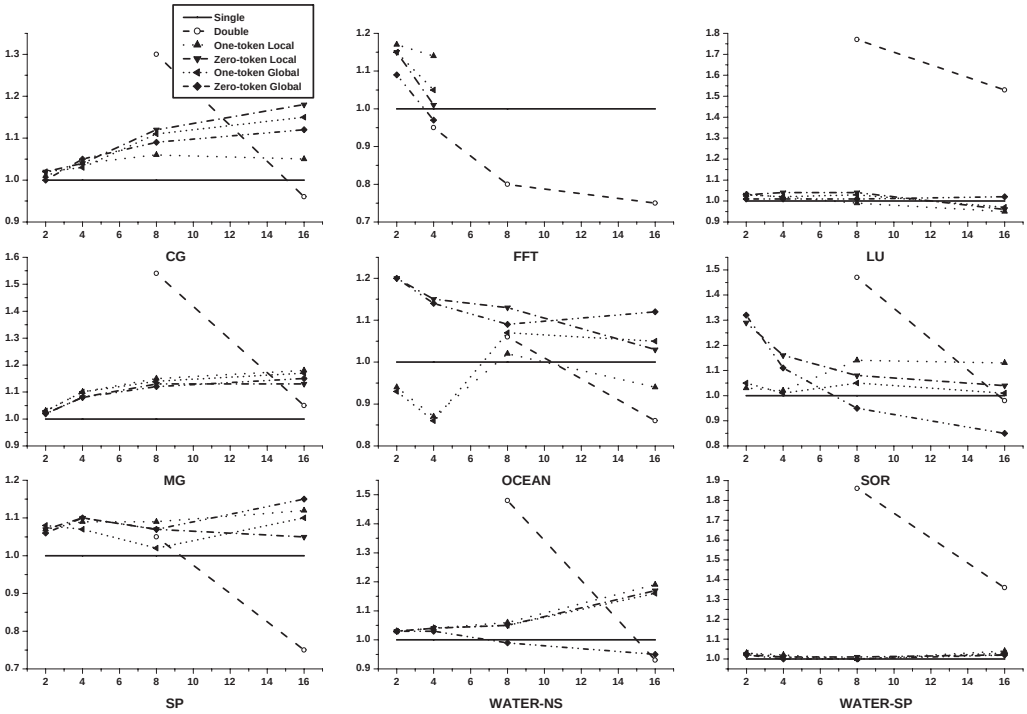


Figure 5. Speedup of slipstream and double modes, relative to single mode. For slipstream mode, four different types of A-R synchronization are shown.

performance of FFT degrades for this data set at 8 CMPs and higher.

There is no consistent 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 6 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. For SP and MG, slipstream mode decreases barrier time, because it 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,

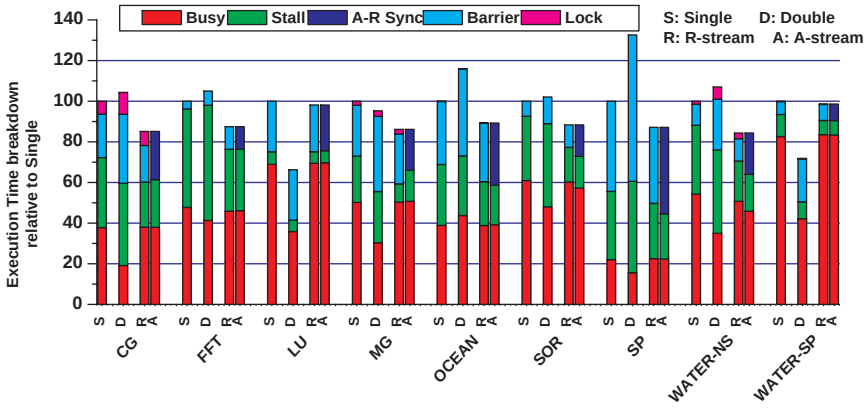


Figure 6. Execution time breakdown for single (S), double (D), and slipstream modes (A ,R), relative to single mode. The best A-R synchronization method is used for slipstream mode.

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.

Figure 7 shows the breakdown of shared data memory requests 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 A-Only 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 7 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 prefetches.

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.

R-stream requests to memory are the result of lines that are invalidated or evicted, and lines that are not referenced at all 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 – in this case, 98% of read requests and 77% of the exclusive requests are for data that is referenced by both streams.

To summarize, 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

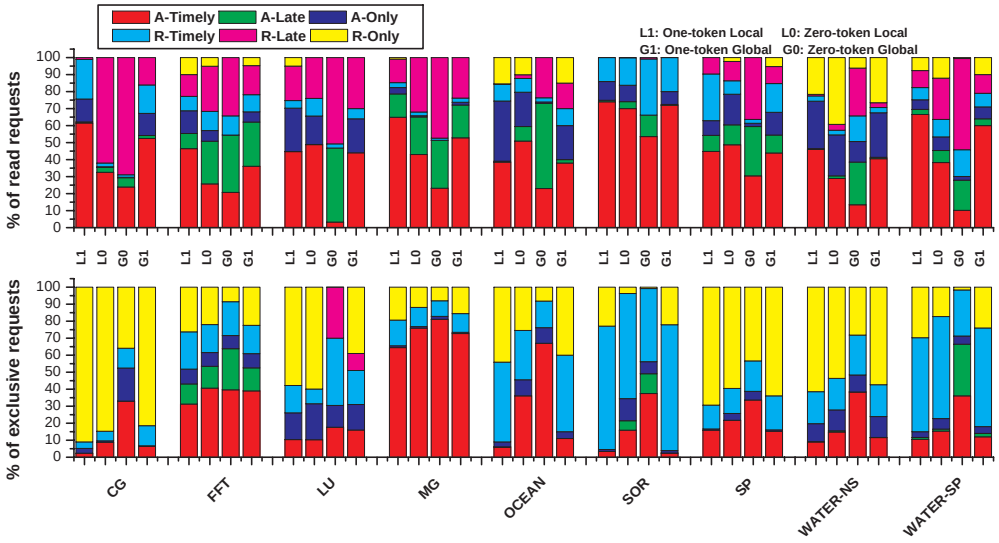


Figure 7. Breakdown of memory requests for shared data.

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 the next section.

4. Prefetching with 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)* [11,12] 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 reduces invalidation messages and writeback requests. A subsequent load from another processor will find the data in memory, without having to request it from the owning cache. A subsequent store will acquire an exclusive copy from memory without having to invalidate copies in other caches.

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.

4.1 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.

4.2 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 8.

In the left half of the figure, the memory directory

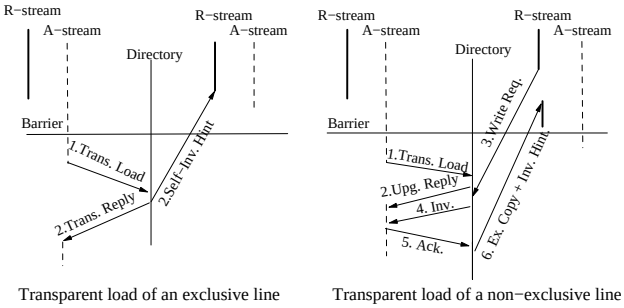


Figure 8. Slipstream-based self-invalidation.

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 right half of Figure 8, 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 [12], lines marked for self-invalidation are processed when the R-stream reaches a synchronization point. Unlike their approach, 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 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 [11] 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 that capability in future work.

4.3 Performance of transparent loads and SI

To evaluate the performance of transparent loads and SI, we focus on the 16-CMP configuration for all applications except for FFT (4 CMPs). 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.

Figure 9 shows the percentage of A-stream read requests issued as transparent 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

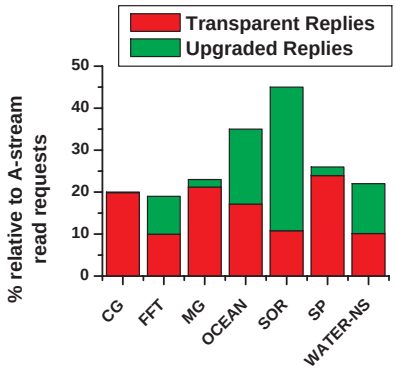


Figure 9. Transparent load breakdown.

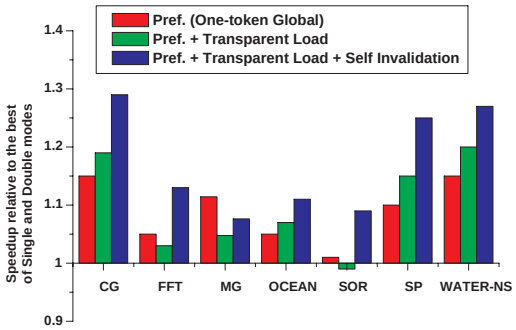


Figure 10. Performance with transparent loads and SI.

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.

Figure 10 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 7), which indicates there are few dependency violations, while about 21% of A-stream requests are handled transparently when self-invalidation is enabled (Figure 9). 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. Compared to the best prefetching configuration for each benchmark (from Section 3), 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. Related Work

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 [3], by the programmer or compiler [13,16], or by pre-computation [1,2,4,6,14,20, 21,25,28]. Slipstream does not require customized hardware tables to guide prefetching. Instead, it harnesses existing processors if increasing the number of parallel tasks is ineffective. It also does not require major programming effort, and exploits run-time information that compilers cannot.

Pre-computation uses helper threads to compute the addresses of problem loads (loads that frequently miss in the cache) ahead of the full program [1,2,4,14,20, 21,25,28]. Problem loads are explicitly identified and targeted, through profiling [21,28] or dynamic identification mechanisms [5,20]. Then, the computation slices that generate the addresses of problem loads are extracted from the full program either manually [28], by the compiler [14], or by hardware mechanisms [5]. 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 [5]. In contrast, slipstream only requires executing a redundant copy of each task (A-stream). A processor running an A-stream does not explicitly pinpoint problem loads, extract their pre-computation slices, continuously fork threads, or micro-manage timing. Timing is managed at a high level, via a one-time choice of A-R synchronization method. An A-stream "gets ahead" simply by skipping synchronization and by not committing shared-memory stores, which does not generally affect forward progress and address generation in parallel scientific applications.

The decoupled access/execute (DAE) architecture [24] 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.

Both DAE and helper threads operate in the context of sequential programs. Therefore, these forms of pre-fetching specialize in moving data between a single processor and main memory. They are not tailored to coordinating communication among distributed processor caches. Reducing memory latency in a multiprocessor setting requires new mechanisms like slipstream's trans-parent load for conveying hints among distributed tasks.

Dynamic self-invalidation (DSI) [12] 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, an aspect this paper borrows from. Last-touch prediction [11] 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. DSI [12], last-touch prediction for SI [11], and other methods for accelerating communication [10,17] all use history to predict future sharing patterns. Slipstream execution mode enables the use of program computation to predict future sharing patterns.

This paper borrows from the slipstream paradigm [26]. Slipstream in a uniprocessor context targets different types of programs and overheads than in a multiprocessor context. In both cases, a persistent redundant copy of the program or task (respectively) is utilized, but A-stream creation, shortening, and recovery, as well as A-stream to R-stream information passing, differ in fundamental ways due to the different target architectures.

6. Conclusions and Future Work

Slipstream execution mode in a CMP-based multi-processor 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 increase task-level concurrency. In this paper, we have introduced a method for creating a shortened A-stream by skipping synchronization and 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 the best of running one or two tasks per CMP on systems with 16 CMPs. Slipstream execution mode requires only moderate, uncomplicated changes to hardware and software. It is selectively applied (used only when needed) and does not inhibit conventional modes of execution.

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.

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 varying the scheme dynamically during program execution.

For the slipstream-based optimizations presented in this paper, information from the A-stream is conveyed to the R-stream through the shared L2 cache and indirectly through memory directories. Further optimizations are possible if the A-stream is able to pass more explicit information about patterns of access to shared data. We view slipstream as a unifying prediction methodology that can address many optimizations. In future work, we will complete the design of an efficient mechanism to explicitly convey access pattern information from the A-stream to the R-stream, and we will apply that mechanism to a variety of multiprocessor optimizations.

7. References

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

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

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

[4] J. Collins, et al. "Speculative Precomputation: Long-range Prefetching of Delinquent Loads," *28th Int'l Symp. on Computer Architecture*, July 2001.

[5] J. Collins, et al. "Dynamic Speculative Precomputation," *34th Int'l Symp. on Microarchitecture*, Dec. 2001.

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

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

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

[9] J. Kahle. "Power4: A Dual-CPU Processor Chip," *Microprocessor Forum*, Oct. 1999.

[10] S. Kaxiras and J. R. Goodman. "Improving CC-NUMA Performance Using Instruction-Based Prediction," *5th Int'l Symp. On High-Performance Computer Architecture*, Jan. 1999.

[11] A. Lai and B. Falsafi. "Selective, Accurate, and Timely Self-Invalidation Using Last-Touch Prediction," *27th Int'l Symp. on Computer Architecture*, June 2000.

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

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

[14] C.-K. Luk. "Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors," *28th Int'l Symp. on Computer Architecture*, June 2001.

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

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

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

[18] K. Olukotun, et al. "The Case for a Single-Chip Multiprocessor," *7th Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems*, Oct. 1996.

[19] M. Rosenblum, E. Bugnion, S. Devine, and S. Herrod. "Using the SimOS Machine Simulator to Study Complex Computer Systems," *ACM Trans. on Modeling and Computer Simulation*, 7(1), Jan. 1997.

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

[21] A. Roth and G. S. Sohi. "Speculative Data-Driven Multithreading," *7th Int'l Conf. on High-Performance Computer Architecture*, Jan. 2001.

[22] T. Sato, K. Ohno, and H. Nakashima. "A Mechanism for Speculative Memory Access following Synchronization Operations," *14th Int'l Parallel and Distributed Processing Symp.*, April 2000.

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

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

[25] Y. H. Song and M. Dubois. "Assisted Execution," Tech. Report CENG-98-25, Department of EE Systems, University of Southern California, Oct. 1998.

[26] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. "Slipstream Processors: Improving both Performance and Fault Tolerance," *9th Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems*, Nov. 2000.

[27] S. Woo, et al. "The SPLASH-2 Programs: Characterization and Methodological Considerations," *22nd Int'l Symp. on Computer Architecture*, June 1995.

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

[29] http://www.nas.nasa.gov/NAS/NPB/