Extending OpenMP to Support Slipstream Execution Mode

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

Abstract

OpenMP has emerged as a widely accepted standard for writing shared memory programs. Hardware-specific extensions such as data placement are usually needed to improve the scalability of applications based on this standard. This paper investigates the implementation of an OpenMP compiler that supports slipstream execution mode, a new optimization mechanism for CMP-based distributed shared memory multiprocessors. Slipstream mode uses additional processors to reduce communication overhead, rather than to increase parallelism.

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.

We extended the Omni OpenMP compiler to generate binaries that support slipstream mode, and we show the performance of slipstream-enabled codes using OpenMP codes from the NAS Parallel Benchmark suite, running on the SimOS simulator. Our extension to OpenMP allowed the benchmarks to achieve an average performance improvement of 14% with static scheduling. For dynamic scheduling the performance improvement is 12% on average.

1. Introduction

OpenMP [3] is a directive-based standard for shared-memory parallel programming. It allows simple incremental parallelization of applications by identifying loops and other regions of code that can be computed in parallel. OpenMP does not provide facilities to control data locality or coherence, as these features are platform dependent. Portability of OpenMP applications puts the burden on compilers and hardware to achieve good performance.

[*]This work supported in part by the NSF Computer Systems Architecture program, contract CCR-0105628.

While a compiler can do analysis to remove unnecessary synchronization and to optimize for locality of data accesses, the overhead of parallelization *vs.* the 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. Likewise, the upper limit of parallelization for decent performance is dependent on runtime information, such as the problem size and the underlying architecture. For this reason, the OpenMP standard includes environment variables to facilitate changing decisions about scheduling and parallelism at runtime.

Other researchers have proposed extensions to OpenMP to express architecture specific optimizations, such as data distribution directives for CC-NUMA [6] and software-DSM [15] 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. In this spirit, we present extensions and compiler support needed to exploit *slipstream execution mode*—a new performance enhancement for multiprocessors built from dual-processor CMPs (chip multiprocessors) [9].

Slipstream execution mode is based on the observation that adding more computational resources does not always reduce execution time for a fixed-size problem. As the problem is divided into smaller pieces to increase parallelism, communication and synchronization overheads begin to dominate or even overtake the reduced computation time. When this occurs, it may be more effective to apply additional resources to reduce communication overhead, rather than to increase parallelism.

Slipstream execution mode considers cache-coherent distributed shared memory (DSM) multiprocessors built from dual-processor CMPs with shared L2 cache, such as the IBM Power-4 CMP [10]. 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. By running ahead, the reduced version prefetches data into the shared L2 cache for use by the true task. In addition, the reference stream of the reduced task represents a view of the future that can be used for coherence optimizations, such as self-invalidation. Ibrahim et al [9] show that slipstream execution mode can reduce execution time by 12-29%, compared to using one task or two tasks per dual-processor CMP.

This paper addresses the problem of supporting slipstream execution 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 (discussed in Section 2.2) 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.

OpenMP allows an application developer to experiment with different scheduling mechanisms and parameters. Both static and dynamic scheduling algorithms are available. Slipstream mode co-exists transparently with both types of scheduling, with no additional effort required by the programmer or user. Our slipstream-enhanced compiler deals with all of the coordination between tasks needed during scheduling.

Slipstream execution mode provides an additional opportunity for extending the scalability of an application. With OpenMP, this opportunity can easily be explored, even for programs that were not written with slipstream in mind. This combination is a powerful tool that frees programmers from system-specific details, while at the same time providing runtime control and selection of the optimal execution mode for a particular combination of system architecture, application, and problem size.

We begin with a brief overview of slipstream execution mode in Section 2. Section 3 describes the requirements for supporting slipstream mode for OpenMP. Section 4 describes the OpenMP compiler used for this study and how it is extended to support this mode. Section 5 describes the simulation methodology and performance results. We show that OpenMP with slipstream mode provides a performance gain of 14% compared with running one or two tasks per CMP. With dynamic scheduling the average performance gain is 12%. Related work is discussed in Section 6.

2. Overview of Slipstream Execution Mode

The motivation behind slipstream execution mode for multiprocessors is to use the available computing resources in the most effective way to reduce execution time. With two processors per CMP, the natural approach is to divide the problem into 2N tasks for N CMPs, assigning one task per processor. As the number of CMPs increases, however, many fixed-size applications will reach a point at which applying additional computational resources will not reduce execution time, due to the overheads caused by communication and synchronization.

Instead of dividing the computation into smaller pieces, slipstream execution mode uses the second processor on each CMP to reduce communication overhead. Each CMP is assigned a single parallel task, which is executed redundantly by two processes. The first process, called the *R-stream*, executes the original task. The second (speculative) process, called the *A-stream*, shortens the task by skipping synchronization events and stores to shared variables.

Running in slipstream mode reduces the latency of memory accesses for the R-stream, because the corresponding A-stream prefetches data into the shared L2 cache. It can also be used to give hints about future behavior of the programs that reduce the latency of data migration. This can be achieved by sending self-invalidation hints to producers of data based on future references by consumers.

2.1. Slipstream View of Shared vs. Private Variables

Slipstream execution mode capitalizes on the fact that control flow and address generation rely mostly on private variables in shared memory parallel applications. This means that identical threads, sharing the same task ID, will generate nearly identical memory reference traces for data associated with shared variables. Shared variables are primarily used to hold application data, but they are also used for scheduling and synchronization. Slipstream requires identifying these parts to be specially handled by threads in slipstream mode. For example, synchronization routines should be skipped by the A-stream and they are also used for synchronization with the R-stream, as will be discussed later. Scheduling (especially non-static) relies on shared variables to make decisions, and the assigned jobs depend on timing. This scheduling code requires also a special handling so that the A-stream and the R-stream get assigned to similar jobs.

An earlier proposal [9] for implementing slipstream faced a problem in identifying shared addresses and scheduling regions. Many parallel programming models do not expose this information explicitly to the compiler. Application programs with embedded dynamic scheduling need to be explicitly modified to run in slipstream mode. This limited the ability to automate running applications in slipstream mode and made their transparent applicability selective. In summary, without explicit identification of shared data, synchronization and scheduling sections, programmer intervention may be needed.

OpenMP requires shared data to be exposed explicitly to the compiler. It also eases the identification of synchronization and scheduling code, as they are mostly implemented by the compiler and not by the programmer. The programmer shows his intent (what to do) to the compiler, and the compiler takes over the implementation part (how to do it). This delineation between and *how* and *what* allows applying slipstream model unconditionally.

2.2. Synchronization between Slipstream A-stream and R-stream

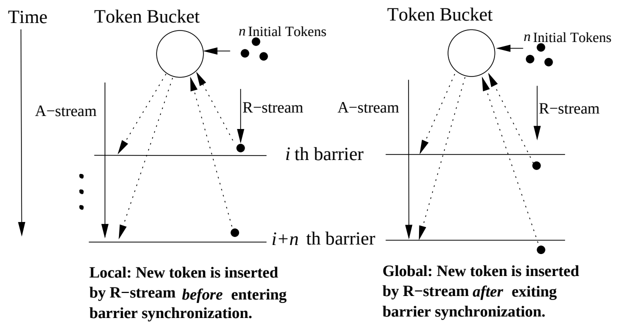
Synchronization between the A-stream and the R-stream serves two purposes. First, it controls how far ahead an A-stream can be ahead of of its R-stream. Second, it is used to check if divergence of A-stream happens and to invoke recovery routine if needed.

Figure 1 shows how synchronization between both streams is controlled. At the beginning of a parallel region, a number of tokens is allocated that controls how far the A-stream can be ahead of its R-stream. The A-stream consumes one token to be able to skip the barrier synchronization, while the R-stream inserts a new token each time it reaches a barrier. The R-stream can insert a token either when it enters the barrier, thus synchronizing with its A-stream *locally*, or before exiting the barrier, thus making its A-stream *globally* synchronized. If the number of tokens allocated is exhausted by A-stream, it waits until a new token is available. The R-stream also can check if its A-stream has reached the same barrier by comparing the number of tokens to the initial value. If it is less than the initial value, the R-stream predicts that its A-stream has visited this barrier. Otherwise, the R-stream can invoke a recovery routine for its deviating A-stream. This synchronization can be implemented using a shared register (or memory location) between the two processors in a CMP. Thus, there are two ways to control A-R synchronization: the number of tokens, and the insertion point of the tokens (local *vs.* global).

A similar synchronization semaphore is used for system input functions and some of the system routines. The synchronization semaphore used with system calls is initialized to zero and the token is inserted by the R-stream when exiting these routines.

3. OpenMP Support for Slipstream Mode

As discussed earlier, there is a runtime component to achieve a good performance that is not easily captured by the compiler. Slipstream execution mode is a runtime mode that can achieve good performance when the overheads of parallelization start dominating the execution time. It is well known that there is a limit to which a problem can attain better performance by increasing the degree of parallelism.



Time
Token Bucket
n Initial Tokens
A−stream
R−stream
i th barrier
i+n th barrier
Local: New token is inserted by R−stream before entering barrier synchronization.
Token Bucket
n Initial Tokens
A−stream
R−stream
Global: New token is inserted by R−stream after exiting barrier synchronization.

Figure 1. A-stream allowable progress is controlled by the number of initial tokens and when new token is inserted.

Slipstream execution mode can provide an additional option in executing a certain parallel region. Different parallel regions may have different amounts of work. It may be advisable for a certain parallel region not to exceed a certain degree of parallelism, while globally we may need a higher number of processors to achieve good performance. Slipstream provides a dimension to use the dedicated resources in an alternative manner. The decision is done per parallel region.

3.1. 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 dedicated has a unit of CMP node. 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 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 A-stream R-stream 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. 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 them by an A-stream that is not paired with the R-stream that will execute them 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 these sections.
  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 optimization 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 these sections 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.

3.2. 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.

3.2.1 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.

3.2.2 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 for slaves.

This decision of scheduling parameters (e.g., chunk size) 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. Considering 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 [16] 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 an additional synchronization overhead, as the scheduling decision should 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. This scheduling decision works as an additional synchronization point between both stream. This may disallow other slipstream optimization, for example self-invalidation. This behavior can be desirable if the amount of work between two barrier is very large compared with the available cache that 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.

3.3. Slipstream Directives

To support slipstream mode the following directive is needed:

!$OMP SLIPSTREAM([type] [, *tokens*])

The *tokens* specify the initial tokens count for synchronization, as shown in Figure 1. 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 assumed 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 disable 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 limit the use of slipstream when the number of CMPs involved in solving the problem exceeds a certain limit.

4. Slipstream-aware OpenMP Compiler

In this work, we extended the Omni OpenMP [11] compiler to support slipstream execution mode. This compiler is freely available [2]. The compiler is supposed to work on IRIX 6.5 environment. We modified it to run on IRIX 5.3, available for our simulation environment.

4.1. 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 spin (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 is an optimizing compiler for OpenMP. 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, coherence overhead, and improve data locality.

4.2. 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 directive into a multi-threaded C program with runtime libraries calls. The compilation process involves: a) parsing source code into an intermediate code, called Xobject code; b) data flow analysis and optimizations are performed by a java class library, called Exc java tools; c) Exc java tool generates a C program with runtime library calls; d) finally, a native C compiler compiles and links the generated file.

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. Specifically, we modified the library for synchronization between threads forming A-stream R-stream pairs, constructs that control handling of parallel constructs, reduction variables 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

Table 1. Simulated System Parameters

CPU	
MIPSY-based CMP Model	Clock Speed: 1.2 GHz
L1 Caches (I/D)	**L2 Cache (Unified)**
Size: 16 KB Associativity: 2 Hit Latency: 1 cycle	Size: 1 MB Associativity: 4 Hit Latency: 10 cycles
Memory Parameters[a] (ns):	
BusTime: 30	NILocalDCTime: 60
PILocalDCTime: 10	NetTime: 50
NIRemoteDCTime: 10	MemTime: 50

[a]Detailed description of these parameters is found in SimOS documentation[17].

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.

5. Simulation Methodology

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 [17], with IRIX 5.3 and a MIPSY-based CMP model. Table 1 shows the simulated machine parameters, including the memory and network latency parameters. 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 2. These benchmarks are an OpenMP port of NAS Parallel Benchmarks 2.3[1], done by the Omni compiler project [2]. All simulations are done on a machine composed of 16 CMPs. For this machine size, the problem sizes serve the purpose of studying the performance when the communication starts to dominate execution time and also to achieve a reasonable simulation time.

5.1. Slipstream Performance with Static Scheduling

The performance measurement for the remainder of the paper will be speedup normalized to single-mode execution

Table 2. OpenMP NPB2.3 benchmarks used in this study and their problem sizes

benchmark	Description
SP	3D Multi-partition for uncoupled systems of linear equations {scalar pentagonal} (24×24×24).
LU	Lower Upper symmetric Gauss-Seidel (33×33×33).
BT	3D Multi-partition for uncoupled systems of linear equations {block tridiagonal} (24×24×24).
MG	Multigrid solver for Poisson equation (32×32×32).
CG	Conjugate Gradient (1400).

(one task per CMP with one processor idle). We assumed that all the parallel regions execute in the same mode (single, slipstream, or double) and the same synchronization (if slipstream is activated).

Figure 2 shows the performance of slipstream and double-mode (2 tasks/CMP) over single-mode (1 task/CMP) execution. For slipstream mode, two 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 global (G0), which allows the A-stream to enter the next session when its R-stream exits the same synchronization event. Figure 2 also shows the execution time breakdown. The time categories are busy cycles, memory stalls, and two kinds of synchronization lock and barrier. Parallelization overheads are also shown as scheduling time, and job wait time. Job-wait time represents the time a process waits for a job to be assigned.

The best performing slipstream gives a performance advantage over the best of single and double mode that ranges from 5% for LU to 20% for MG (13.5% average). Obviously, for fewer number of CMPs, running in double mode can yield better performance compared with single and slipstream. We focused on the region (degree of parallelism) where these benchmarks benefit more from reducing the communication overheads.

The choice of A-R synchronization affects the prefetching behavior and consequently the performance. Figure 3 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 *A-Only* component is considered harmful, as it reflects an unneces-

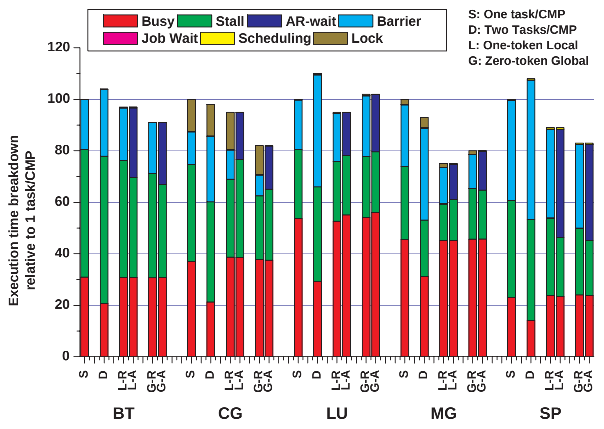


Figure 2. Execution time breakdown for single, double and slipstream modes (static scheduling).

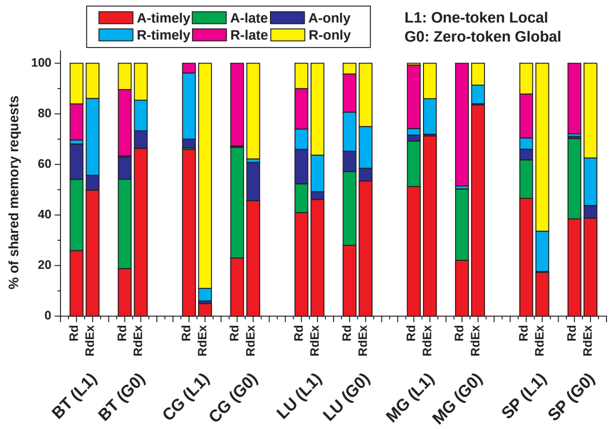


Figure 3. Breakdown of memory requests for shared data (static scheduling).

sary 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*. Exclusive requests by A-stream are due to converting some of the shared stores into prefetches. This conversion occurs only when the A-stream is in the same session with its R-stream and no resource contention exists.

Zero-token global exhibits average A-timely read requests of 26% compared with 46% for one-token local. Zero-token global has also a higher average of late read requests (34% compared with 15% for one token local). This is because the A-stream is not allowed to run very far ahead of its R-stream. On the other hand, zero-token global has a higher read exclusive coverage (58% compared with 38% for one-token local). Zero-token global has also less premature prefetches (3% compared with 8% for one-token local).

Prefetching performed by R-stream is due to two reasons. First, the A-stream is behind his R-stream, which may happen with tight synchronization (G0). This is expected, especially with the use of operating system (IRIX) that does not recognize slipstream mode where A-stream and R-stream are scheduled and serviced independently. Second, the R-stream is requesting data that were evicted or invalidated after being prefetched by A-stream, which may happens with loose synchronization (L1). It is notable that the R-stream wait-synchronization for its A-stream is negligible for all applications and synchronizations, which shows that the A-stream is mostly ahead of his R-stream.

Individually, each application has a tendency to favor one synchronization scheme over the other. CG, LU, and MG favor the loose synchronization of one-token local, while BT and SP favor the conservative synchronization provided by zero-token global. Other components of shared data requests complete the images about how correlated the memory references between A-stream and R-stream. For example, with zero-token global, 95% of the shared data read from memory is referenced by both streams and 70% of the shared data requested exclusively are referenced by both streams. Under static scheduling, the scheduling component of time is negligible as shown in Figure 2.

The performance gained, in this section, requires a simple addition of slipstream directive to the source code. We changed the synchronization method as well as activating/deactivating slipstream at runtime while using the same binary. The results presented shows the sensitivity of performance to the type of A-R synchronization. This encourages further exploration to select different A-R synchronization for different parallel regions.

5.2. Slipstream Performance 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 operation and not on skipping synchronization. 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 potential lack of cache affinity.

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. The benchmarks (except for CG) contain a small number of coarse-grained parallel loops. So, we used the compiler defaults for all applications, except for CG, where we used chunk size equal to half 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.

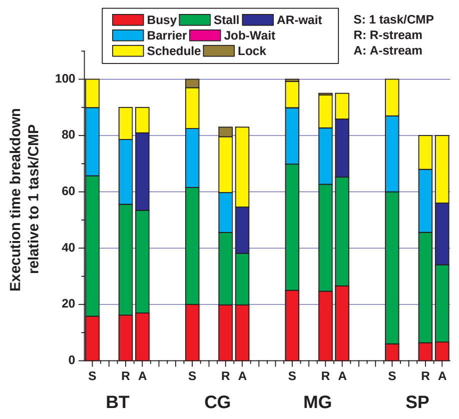


Figure 4. Execution time breakdown for single, zero-token global slipstream mode (dynamic scheduling).

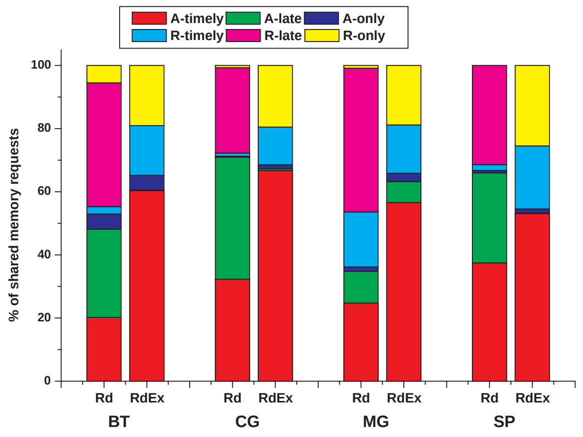


Figure 5. Breakdown of memory requests for shared data (dynamic scheduling). Slipstream synchronization is zero-token global.

The second property is the possible data migration due to these scheduling policies.

We used the same benchmarks used in the previous section except for LU as static scheduling is programmatically specified in this benchmark for a significant portion of the code. All other benchmarks do not specify a scheduling policy.

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 4 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 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.

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.

6. Related Work

The OpenMP standard does not include directives that specify hardware-specific optimizations. Although this approach maximizes the portability of codes written with this standard, the desire for performance inevitably leads to non-compliant extensions that target specific hardware issues. Data distribution specially for CC-NUMA architecture [6] and Software-DSM [15] (distributed shared memory machines) were introduced to cover the gap intentionally left in the standard. Other extensions [12, 14] are proposed to improve execution of numerical codes. Extending OpenMP to support slipstream mode complements these studies by devising an optimization for CMP-based multiprocessor systems.

Slipstream execution mode for multiprocessors is closely related to the slipstream uniprocessor paradigm [19]. 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. Uniprocessor slipstream is compute-centric, geared toward reducing the execution time of a sequential program. Its A-stream is reduced by speculatively removing instructions that do not affect the outcome of the program. The A-stream passes values and branch outcomes directly to its companion R-stream. In the multiprocessor domain, the A-stream is shortened by simply skipping long-latency events (synchronization and shared stores). The A-stream and R-stream primarily communicate indirectly, through the shared L2 cache and the directory-based coherence protocol.

Both approaches are related to other mechanisms that use multiple threads to tolerate long latencies and prefetch data in uniprocessor systems [4, 5, 7, 8, 13, 18, 20]. Slipstream execution mode, however, does not require microarchitectural support for threads, does not explicitly identify problem loads and their pre-computation slices, and does not continuously fork threads or micro-manage timing.

7. Conclusions

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 optimization done 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. The proposed extension gives an average performance gain of 14% for five benchmarks from NAS NPB with static scheduling of the code.

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 code can benefit both programmers and end users, improving the performance of portable parallel applications.

References

[1] NAS Parallel Benchmarks. http://www.nas.nasa.gov/NAS/-NPB.

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

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

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

[5] 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.

[6] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C. A. Nelson, and C. D. Offner. Extending OpenMP for NUMA machines. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 48. IEEE Computer Society Press, 2000.

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

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

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

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

[11] K. Kusano, S. Satoh, and M. Sato. Performance Evaluation of the Omni OpenMP Compiler. *Lecture Notes in Computer Science*, 1940:403, 2000.

[12] J. Labarta, E. Ayguadé, and J. Oliver. New OpenMP Directives for Irregular Data Access Loops. *Second European Workshop on OpenMP*, Sept. 2000.

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

[14] L. Meadows. Extending OpenMP to Improve Scalability for Numerical Codes. *Workshop on OpenMP Applications and Tools*, Aug. 2002.

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

[16] D. Nikolopoulos, E. Artiaga, E. Ayguadé, and J. Labarta. Exploiting Memory Affinity in OpenMP through Schedule Reuse. *3rd European Workshop on OpenMP (EWOMP'01)*, Sept. 2001.

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

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

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

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