CACHE AND INTERCONNECT ARCHITECTURES IN MULTIPROCESSORS

edited by

Michel Dubois
University of Southern California

and

Shreekant S. Thakkar
Sequent Computer Systems

KLUWER ACADEMIC PUBLISHERS
Boston/Dordrecht/London

$$+\left[\frac{(J-1)\cdot W\cdot(1-W^2)}{(J-1+W)\cdot(1+(J-1)W)}+\frac{(J-1)\cdot W^2\cdot(1-f)}{J-1+W}\right]\cdot t_1$$
$$\left.+\frac{(J-1)\cdot W^2\cdot(1-fW)\cdot(J^2+2JW-2J-2W+2)}{(J-1+W)^2\cdot(1+(J-1)W)}\cdot t_2\right\},$$

re $t_1=\max(t_{word}, t_{inv})$ and $t_2=(t_{mc}-t_{cc})$, if $t_{mc} > t_{cc}$, or $t_2=0$, other-
e.

Synapse

$$\lambda_{total} = \frac{1}{ls}\left\{\frac{(J-1)\cdot W^2}{J-1+W}\cdot t_{cc}+\frac{(J-1)\cdot W\cdot(JW-2W+J+2)}{(J-1+W)\cdot(1+(J-1)W)}\cdot t_{mc}\right.$$
$$\left.-\frac{2\cdot(J-1)\cdot W^2\cdot f}{J-1+W}\cdot t_{mc}\right\}.$$

Illinois

$$\lambda_{total} = \frac{1}{ls}\cdot\left\{\frac{(J-1)\cdot W}{1+(J-1)\cdot W}\cdot t_{cc}+\frac{(J-1)\cdot W\cdot(1-W\cdot f)}{J-1+W}\cdot t_2\right.$$
$$+\left[\frac{(J-1)\cdot W\cdot(1-W^2)}{(J-1+W)\cdot(1+(J-1)W)}\right.$$
$$\left.\left.+\frac{(J-1)\cdot W^2\cdot(1-f)}{J-1+W}\right]\cdot t_{inv}\right\},$$

re $t_2=(t_{mc}-t_{cc})$, if $t_{mc} > t_{cc}$, or $t_2=0$, otherwise.

Berkeley

$$\lambda_{total} = \frac{1}{ls}\cdot\left\{\frac{(J-1)\cdot W}{1+(J-1)W}\cdot t_{cc}+\frac{(J-1)\cdot W\cdot(1-W^2)}{(J-1+W)\cdot(1+(J-1)W)}\cdot t_{inv}\right.$$
$$\left.+\frac{(J-1)\cdot W^2\cdot(1-f)}{J-1+W}\cdot t_{inv}\right\}.$$

Performance of Parallel Loops using Alternative Cache Consistency Protocols on a Non-Bus Multiprocessor

Russell M. Clapp   **Trevor N. Mudge**
Advanced Computer Architecture Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109-2122

and

James E. Smith
Cray Research, Inc.
900 Lowater Road
Chippewa Falls, Wisconsin 54729

Abstract

In this paper we present the results of a preliminary study of the performance of parallel loops on a non-bus shared-memory multiprocessor. Parallel loops are defined to be "do" or "for" loops whose iterations are independent and can therefore be executed in parallel. These are potentially the greatest source of parallelism in a program and, therefore, it is important to demonstrate that this potential can be realized before exploring other sources of parallelism. The sources of inefficiency that can limit the parallelism are the mechanism for maintaining cache consistency and the algorithm that schedules the loops across the processors. As part of our study we examined the impact on parallel performance of two software and two hardware cache consistency techniques as well as three scheduling policies.

INTRODUCTION

Parallel processing is an increasingly important technique used to speedup the execution of compute-intensive scientific codes. In this paper we consider the use of shared-memory multiprocessors in which the individual processors cooperate to speedup the execution of a single program. In order to avoid re-writing the large base of scientific codes written in sequential languages, methods have been developed to parallelize these programs. One of the most elementary of these methods is to simply execute multiple iterations of "do" loops or "for" loops in parallel. We will refer to these loops as "parallel loops".

Parallel loops can be detected through data dependency analysis performed at compile-time [6, 7]. However, several languages and language extensions have been proposed that contain **doall** and **doacross** style loops, which allow the programmer to explicitly state the parts of the loop that may be executed in parallel [8, 9, 10, 11].

A **doall** loop is one in which each iteration may be executed in any order and in parallel, i.e., there are no data dependencies between iterations. In a **doacross** loop, one or more data dependencies exist between iterations which imposes an order of execution. In this paper we limit our experiments to parallel loops that conform explicitly or implicitly to the **doall** semantics. They represent potentially the greatest source of parallelism in a program, therefore, as a first step, it is important to demonstrate that these loops can be executed in parallel with a high degree of efficiency before exploring other sources of parallelism.

To avoid performance degradation in a shared-memory multiprocessor due to memory congestion it is necessary to include a cache or local memory with each processor and provide a high-bandwidth connection to main memory. Furthermore, techniques must be employed (either in hardware or software) to maintain data consistency across the caches and main memory. Many hardware solutions to the cache consistency problem have been studied, but most are snooping protocols, best suited to shared-bus multiprocessors. Considerably less attention has been given to protocols for non-bus systems as well as software techniques for maintaining consistency.

In this paper, we examine the impact of two software and hardware cache consistency techniques in a non-bus multiprocessor on the performance of parallel loops. We also examine the impact of three scheduling policies for the microtasks that result from parallelizing the loops. We perform our experiments using a high performance register-transfer level simulator of the Astronautics ZS series of multiprocessors. The simulator interprets code from executable binaries. These binaries are created using compilers and an assembler for an existing uniprocessor version of the ZS series, the ZS-1. The sequential code is compiled and then hand parallelized by adding basic run-time system and synchronization code at the assembly language level. The run-time extensions include support for the three scheduling policies, as well as software consistency actions when necessary. Synchronization is performed using a special set of shared "semaphore" registers, which keeps the synchronization traffic separate from the cache and prevent it from skewing the results of the consistency overhead measurements. The use of a simulator for our experiments provides us with the flexibility to alter the consistency hardware, basic cache parameters, and the number of processors.

There have been a number of recent studies of multiprocessor cache performance reported in [1, 2, 3, 4, 5]. The first three of these studies focused on bus-based multiprocessors using snooping protocols for cache consistency. The other two examined the performance of hardware directory schemes and software schemes for cache consistency. The experiments in these two studies were for a "generic" RISC-based multiprocessor and relied on real multiprocessor traces from a collection of application programs. These traces were obtained from a four processor CISC-based multiprocessor and included operating system activity.

The distinguishing characteristics of our study are: 1) the target is an actual design for a multiprocessor system that is an extension of an existing uniprocessor computer, 2) the performance measurements are made by using an interpreter driven register-transfer level simulator, 3) the simulator inputs are real executable binaries of microtasking parallel programs that are constructed using existing compiler/assembler tools, and 4) there is no operating system code included in the measurements, nor is there any need to use the cache or main memory for synchronization purposes. While this approach produces results that are particular to this system and the parallel programs simulated, the results do provide a clear view of events that occur in a microtasking environment on a real multiprocessor system. In particular, the study exposes the difficulties and potential pitfalls in pursuing fine-grained parallelism on a non-bus multiprocessor.

In the remainder of this paper, we provide some background on the issues surrounding this problem area, describe our experimental testbed, and discuss the results of our experiments while offering some conclusions. The next section provides background on the run-time support necessary for parallel loops, directory schemes for hardware supported cache consistency, and software schemes for cache consistency. The following section describes our experimental testbed, including the ZS multiprocessor simulator, the construction of parallel programs for simulation, the implementation of the run-time support code, and the implementation of both software and hardware schemes for cache consistency. In the last two sections we present the results of our experiments and conclude with a discussion of those results.

BACKGROUND

Microtasking Run-Time Support

The run-time system that supports program parallelism is the vital component of the parallel processing system because it sets limits on the system's performance. A straightforward run-time system is sufficient to support loop parallelism. Its simplicity is derived from the restricted form of parallelism provided by loops. Since the multiple iterations of the loop execute with the same local environment (that of the enclosing subprogram), there is no need to set up separate data or stack areas. Further, there is no communication between loop iterations, therefore groups of one or more loop iterations can be scheduled as one *microtask* for an available processor without concern for dependencies between the microtasks. We also restrict parallel loops from being nested, to avoid complicating the microtask run-time support. By eliminating nested parallel loops, the implementation is simplified without a severe reduction in parallelism. Restructuring compilers may coalesce loops to increase parallelism and provide a larger grain size. The grain size must be large enough to mask the run-time support overhead, but not too large that it leads to load imbalance among the processors. Loop restructuring techniques are described in [12]. A complete description of loop types and their properties can be found in [13].

The run-time support for microtasking relies on the *self-scheduling* paradigm [14, 13, 15]. In this paradigm, the multiple processors executing a program access a shared run-queue to obtain units of work, loop iterates in this case. The work in the queue is represented by a record that indicates the address of the code as well as the number of times it is to be executed. In the simple case, the run-queue may only hold one such record at a time, providing for a very simple and efficient implementation. The processors used by the program are dedicated, and may access the shared run-queue without entering the operating system. This allows low overhead scheduling, which in turn provides an opportunity to exploit fine-grained parallelism. These microtasks follow *run-to-completion* semantics which enable the run-time system to schedule them only once before they complete.

In a microtasking system, a program runs serially on a processor until a loop is entered. It then performs a synchronization operation to allow other processors to begin accessing the run-queue. The queue is read and modified so that each processor obtains an increment of the work to be done. The processors continue to schedule more work for themselves until there is none left. When the microtasks finish, they usually synchronize at the point immediately following the loop. However, in some cases, the serial portion of the code following the loop may execute in parallel with some of the parallel microtasks. Any other available processor may also reload the run-queue at this point with data that represents the next loop to be executed in parallel. Data dependencies between the loop, the serial code, and any following loops determine how much overlap is possible.

There are several variants of self-scheduling that are possible. When work is obtained from the queue, a microtask of one or more loop iterations may be selected. The number chosen is referred to as the *chunk* size [13]. It is also possible to vary the chunk size dynamically as multiple processors are obtaining work. One such possibility is *guided self-scheduling* [15]. Another possibility is to compute an "optimal" chunk size based on the number of iterations and the number of processors available. These strategies all attempt to optimize the trade-off between load balancing and overhead. In our experiments, we compare the performance of self-scheduling with a fixed chunk size of 1 (which we simply call "self-scheduling"), guided self-scheduling, and optimal chunk size scheduling. The details of our implementations of these approaches are given in the EXPERIMENTAL TESTBED section below.

Cache Consistency

With the emergence of bus-based shared-memory multiprocessors in recent years, the topic of cache consistency has received considerable attention. The bulk of this attention has been focused on *snooping* cache consistency protocols, which are ideal for bus-based systems. However, these schemes are not well suited to systems which use a non-bus style of interconnect between processors, their caches, and memory because they require the ability to broadcast addresses to each cache and main memory. In fact, several multiprocessors which do not use a bus interconnect do not support cache consistency in hardware (e.g., the Astronautics ZS series, the IBM RP3, and the Evans & Sutherland ES-1). Non-bus multiprocessor systems with caches or "transparent" local memories must rely on alternative cache consistency techniques.

We can categorize software mechanisms for cache consistency into three general types that rely on: 1) non-cacheable data; 2) bypass, write through, and/or flush of cache blocks; or 3) cached write buffers with merging. The first type is the classical technique where shared data is made non-cacheable, typically on a page boundary. The second type requires that the compiler utilize information about data dependencies and alignment of words in cache blocks to bypass the cache on a fetch if necessary, and to write through or flush blocks to main memory when needed. Several implementations to this approach have been proposed [16, 17]. The third type allocates cache blocks for result data areas, and then merges these blocks together at the conclusion of the parallel loop or section to form the final result in main memory. It has been implemented on an existing distributed memory multiprocessor using pages as the write buffer size [18]. In our experiments, we have tested this approach using the minimum number of cache blocks required to represent the entire result area of one parallel loop as the write buffer size.

Directory schemes for cache consistency typically keep information regarding cache block location and modification status in a central location. An early scheme proposed by Censier and Feautrier [19] keeps a directory entry for each potential cache block in memory. The entry contains bits to indicate which caches, if any, possess the block and whether or not the main memory version is up-to-date. Thus, for an N processor system each entry would have at least $N+1$ bits. A modification to this scheme is to restrict blocks to at most one cache and then replace the bits with an index value to indicate the cache where the block resides. In both cases, the directory information is used on cache misses or non-private hits to retrieve the up-to-date cache block and, if necessary, invalidate or update other copies and update the tags and directory according to the specifics of the protocol. A survey of several central directory based schemes can be found in [4].

Synchronization

A synchronization instruction is usually based on an indivisible read-modify-write operation on either a main memory word or special memory hardware. Examples include test-and-set [20], fetch-and-op [21], and compare-and-swap [20]. The advantages of using main memory are generality and scalability. The disadvantages are slowness and added complexity to deal with the interaction between hardware cache consistency and synchronization [22]. Alternatively, special memories or shared registers may be provided for fast synchronization. Their disadvantages are that they are limited in number and they do not readily scale to large numbers of processors [23, 24, 25].

The parallel programs used for our experiments make use of the ZS's shared "semaphore" registers for synchronization. The ZS provides 32 sets of semaphore registers, each set consisting of 32 registers, each 32 bits long. Several fetch-and-op instructions are provided that operate on the semaphore registers. Instructions are also provided for the first 8 registers in each group that enable processors to block until the value of a particular register becomes either positive or negative. Together with the ability to read and write these registers, these instructions make the semaphore registers ideal for holding addresses, implementing barriers, and computing indices for a microtasking run-time system.

EXPERIMENTAL TESTBED

ZS Multiprocessor Simulator

The Astronautics ZS series of computers are based on a proprietary 64-bit processor directed at numeric applications. The processor is heavily pipelined and is capable of issuing two instructions per clock period. Memory accessing and floating point are decoupled by using distinct instruction issuing streams for each. Memory accesses are also buffered using processor queues for integer and floating point loads and stores. These features permit dynamic scheduling between addressing and floating point functions, and successfully hides memory latencies in many cases [26]. This type of architecture is referred to as Decoupled Access/Execute (DAE) [27].

The ZS-1, a uniprocessor, was completed (both hardware and software) and has been operational for some time. The hardware is constructed to support up to 16 processor systems, but multiprocessing software is incomplete. ZS-series multiprocessors use a shared set of registers for low-overhead interprocess communications. The multiprocessing hardware has been checked out and has been used for small test cases. Simulations reported in this paper use accurate timings based on the actual ZS multiprocessor hardware.

The interconnection network in the ZS multiprocessor system is essentially a crossbar network. The data path is four words (256 bits) wide, and is optimized for 16 word (one cache line) transfers. To support references to non-cacheable data, smaller transfers, down to one byte, can be accommodated, but the timing for smaller transfers is the same as for a full 16 word transfer.

The multiprocessor simulator is a register transfer-level simulator and program interpreter. Based on the instruction and address information supplied by the interpreter, the busy times of the functional units, load and store queues, cache, main memory, registers, pipelines, etc. are modeled. These devices are advanced each clock period in accordance with any dependencies that are present. A file of system parameters is also used by the simulator to define cycle time requirements for the functional units, memory access, and the queues. Additional parameters include cache line size and data associativity, functional unit requirements for each instruction, and the processor clock speed. Except where otherwise noted below, the simulations we ran used system parameter values consistent with the ZS-1 hardware. In the uniprocessor case, we found the simulator timing results to be within 5% of actual running time on the ZS-1. The discrepancy is due to the lack of address translation faults in the simulator. In order to avoid different versions of a "warm cache" between the different hardware configurations tested, the data cache for each processor is flushed before execution of the timed loop. The cold cache approach did not alter the results greatly, and provided each test case with identical starting positions.

Parallel Program Construction

Parallel programs were constructed for our experiments from two different versions of a matrix multiply program written in (sequential) FORTRAN. The part that we parallelized and tested were the triply nested do loops that are the kernel of the multiply. These nested do loops can be thought of as triply nested doall loops, since there are no data dependencies between the iterations of the loops. The source code we used for our tests and the assembler code generated by the compiler are discussed in the RESULTS section below.

The programs were compiled on the ZS-1, and then disassembled. At the assembly source level, we added instructions to implement the self-scheduling run-time support. Each processor executes the same code, so the run-time support code is responsible for synchronizing these multiple threads of execution and assuring that each processor executes a unique subset of the total work to be done. The run-time support code added to both parallel programs was written to be independent of the number of processors used to execute it. This code only assumes that a count of the number of available processors is provided in one of the semaphore registers.

We parallelized our test programs by allocating microtasks that executed some subset of the iterations of the outermost doall loop. As stated above, we tested three different dynamic scheduling policies, all based on the concept of processor self-scheduling. The first, which we refer to as chunk scheduling, computes an "optimal" chunk size by dividing the number of iterations for the outermost doall loop by the number of processors available. If the numbers do not divide evenly, an extra "chunk" of the leftover iterations is also created. The second scheme, which we simply call self-scheduling, creates one microtask for each iteration of the outermost doall loop. The third technique, called guided-self scheduling, allocates a number of iterations equal to $\lceil \frac{R_i}{p} \rceil$ where R_i is the number of iterations remaining to be scheduled at step i and p is the number of processors [13, 15].

For chunk scheduling, one processor computes the chunk size and places this result, the starting address, and the maximum iteration value in separate semaphore registers. At the starting address, just before the loop bodies, the beginning iteration value for a microtask is converted (using the chunk size and the iteration limit) into beginning and ending values to be used for the loops (e.g., multiplied by word size to make a proper array index). After reading the semaphore registers and before loop execution, another semaphore register is decremented. When this counter reaches zero, it indicates that the semaphore registers can be reloaded with values pertaining to the next parallel loop for execution. While the code to compute the chunk size and load the semaphore registers is executing, all other idle processors are waiting to enter the scheduling code. When the semaphore register holding the total number of iterations to be executed is written and becomes greater than zero, the waiting processors begin executing a fetch&decrement operation on the register to acquire a unique index for that microtask.

The self-scheduling code is similar but less complex. There is no need to compute a chunk size since the chunk size is always equal to one. Also, there is no need to compute an ending value, since the outermost loop is executed only once. This simplifies the code of the loop body by eliminating the test at the end of the outermost loop that normally would determine whether or not the last iteration for that microtask had been executed.

The code for guided self-scheduling is a little more complex. As part of the process of acquiring a unique index for each microtask, several computations must be made to get the proper chunk size as well as the starting and ending values for the iteration range. Because the global number of iterations remaining is the basis for these computations, they must be performed in mutual exclusion until this global value can be updated. This effectively creates a critical section of several assembly language instructions to perform a self-scheduling operation. In addition to holding addresses and key index values, the semaphore registers are also used to implement a binary semaphore that ensures mutual exclusion for the scheduling operation. While the chunk scheduling approach also requires a chunk calculation, it is performed only once before any microtasks begin. For guided self-scheduling, a chunk calculation must be performed at each scheduling point, thus increasing the scheduling code critical section from one fetch&decrement instruction to a lock acquisition and several integer arithmetic instructions.

Implementing Cache Consistency

In our experiments, we evaluated two software and two hardware consistency schemes. The software schemes consisted of 1) making result data non-cacheable, and 2) using local memory management instructions to merge multiple copies of the result data. The ZS provides mechanisms to make pages of virtual memory non-cacheable as well as several instructions for allocating and flushing cache blocks. The hardware schemes we evaluated are two variants of a central directory based approach to cache consistency. In order to evaluate the hardware schemes accurately, the ZS simulator was modified to incorporate them. These consistency techniques are described in more detail below.

Software Consistency While the first software cache consistency scheme is straightforward, the second warrants some further explanation. This second consistency technique utilized the "allocate block" and "flush block" instructions to manage a local copy of the result data in each cache. Each processor allocated enough blocks to hold the entire result array at a temporary virtual address distinct from that used by the other processors and distinct from the result area. As a side effect of the allocate block instruction, each word in each block is initialized to a value of zero. Each processor then proceeds with its share of the parallel loop iterations, performing one or more scheduling operations depending on the self-scheduling technique used. After all iterations have been scheduled, the first processor to complete its work proceeds by flushing its entire result area to main memory at a virtual address reserved for the result using multiple flush block instructions. Each successive processor to complete its iterations then, in mutual exclusion with other processors, reads each 64 bit word from the result area, performs an "OR" operation between that word and the corresponding word stored at its temporary location, and places the result at the result area's virtual address. After each word is read and updated, the cached result area is flushed to main memory.

An alternative approach to software consistency would be to flush blocks as they are written in the body of the loop. Techniques for this style of consistency have been proposed in the literature [16]. However, due to the fact that loop iterations are allocated dynamically with our run-time support, and that the compiler produced code accesses words in blocks in a non-sequential fashion, we would have to completely rewrite the generated code as well as restrict iteration allocation in order to ensure consistent results. This problem would be simplified if individual words could be flushed instead of blocks, or if the word size and cache block size were the same. However, reducing the cache block size to such a small value would eliminate the positive effects of spatial locality obtained with a multiword block. A policy of prefetching one word blocks may compensate for this problem [28], but this requires sophisticated compilers and architectural modifications and is beyond the scope of our study.

Several properties of the codes we simulated enabled us to use the temporary result area approach to consistency. The most important property is that while any number of processors may need to access the same logical block, no two processors require access to the same word within a given block. This creates a situation where each cache contains a temporary result area with words that contain either zero or a final result, and, no two caches contain a final result in the same logical word. This enables the OR instruction to be used to merge the temporary result areas. This approach to data consistency has been used in various forms in other systems (e.g., the Myrias computer [18]).

If the temporary result areas are too large to remain in the cache along with the other referenced data for the duration of the loop iteration executions, each processor must allocate its temporary result area at a different virtual address. If block replacement in the cache then effects the result area, its values will not be corrupted by collision with other processor's temporary results. This approach may require that a large virtual memory space be available to the parallel program.

Hardware Consistency The hardware cache consistency schemes we evaluated are based on the central directory approach proposed in [19]. While this approach is similar to the one originally proposed by Tang [29], it requires fewer cache tags and a smaller central directory. A survey of directory based cache consistency techniques can be found in [4].

The hardware consistency scheme implemented requires a central directory with an entry for each block. This entry contains several bits that indicate which caches contain the block and whether or not the main memory copy is up-to-date. In one scheme, originally proposed in [19], each directory entry contains one bit for each cache in the system to indicate the block's presence in that cache. There is also one bit for each entry that indicates whether main memory is up-to-date with a cached block. If it is not, this bit is set, and the block may reside in only one cache. This consistency scheme does not require any additional cache tags than those already provided by the hardware. These include a valid bit, a dirty bit, and least recently used bits.

A write miss makes a block dirty and consequently exclusive to that cache. The directory must be consulted to invalidate any other cached entries. A write hit must also consult the directory and propagate invalidations if the the block is not already dirty (and thus private). The modified bit in the directory must also be set when a write to a block is performed. Read hits and dirty write hits may proceed without accessing the directory. A read miss requires a directory update for that cache's presence bit for that block. Read and write misses must also supply the up-to-date value of the block to the requesting cache. If the modified bit is set in the directory on a cache miss, main memory must read the block from the cache containing the up-to-date value and clear the associated dirty bit before supplying the data to the requesting cache.

A variation of this scheme proposed in [4] was also tested. Instead of providing a presence bit for each cache in each directory entry, only enough bits are provided to encode an index to one cache in the system. This scheme provides exclusive access for each cached block, and requires an invalidation whenever a processor accesses a block that is cached elsewhere in the system.

As mentioned above, our experiments for evaluating hardware cache consistency techniques required some modifications to the simulator, effectively changing the architecture of the ZS. We also made several simplifying assumptions to avoid a detailed redesign of the hardware. The first assumption is that no race conditions exist between checking local cache tags and accessing the central directory. The simulator updates the cache tags, the directory, and performs invalidations at the point of the cache miss. The processor in this case then idles the required number of cycles to simulate the time taken to perform these updates. Any subsequent accesses by other processors always see the most up-to-date cache tags and directory values. Because individual words are not shared in our test programs, subsequent block accesses by other processors need not wait for previous ones to complete before updating cache tags and directory values. The first processor will cache the block and load the referenced word into the load queue, but the block will no longer be valid.

Although cache tags and directory entries are updated instantaneously, processors must wait additional cycles before accesses are complete if the requested memory bank is busy or the block to be accessed is dirty in another cache. We assume that the directory is interleaved across the memory banks so that the block requested resides in the same memory bank as its directory entry. This provides mutual exclusion for directory entries, since only one processor may access a given memory bank at any one time. We also assume that the directory can be updated, any necessary invalidations can be sent, and main memory can be read all in the time it takes to perform a main memory access. If the requested memory block is dirty in another cache, we assume that memory can be updated and the value supplied in one additional main memory access time. Since the requesting processor has control of the main memory bank when the modified bit is checked, we do not queue the request to write the up-to-date cache block back to main memory and update the directory.

The simulator provides interlocks and memory bank arbitration so that banks are accessed in mutual exclusion. Bank conflicts will add delays to the completion of requests, and these times are effectively added to the base memory access times we assume for directory accesses. While our assumptions about race conditions, the ability to send invalidations, the ability to read up-to-date blocks in other caches, and the ability to add a directory to the memory system most certainly simplify our experiments, we believe that these tests do reflect the additional delays brought about by cache misses due to invalidations and memory accesses to modified blocks.

RESULTS

Source Code

The programs we used in our experiments are two versions of a double precision floating point matrix multiply. The basis for these programs is kernel 21 of the Livermore FORTRAN Kernels [30]. We produced two versions of generated code for this kernel by interchanging the order of the do loops. In order to encourage the compiler to unroll inner loops and to make parallelization of the program easier, we produced the first version of the code by making the innermost do loop into the outermost do loop. This resulted in the following FORTRAN code:

```
      dimension PX(25,101), CX(25,101), VY(101,25)

      do 15 j = 1, n
      do 15 k = 1, 25
      do 15 i = 1, 25
         PX(i,j) = PX(i,j) + VY(i,k) * CX(k,j)
   15 continue
```

This version produced the fastest code in the sequential case for several reasons. In addition to enabling the compiler to unroll the innermost loop, the indexing patterns in this case produced a unit stride in each of the three arrays. With the large (128 byte) cache line size, this version warmed the cache quickly and produced a minimum of cache misses. Because the innermost loop accesses a different element of the result matrix on each iteration, the compiler did not accumulate partial results in registers. This did not degrade performance, however, as the low number of cache misses combined with the ZS's dual instruction issue capability enabled the average number of cycles per instruction to approach 0.6.

Although the version of the code shown above is the fastest in the uniprocessor case, its heavy use of memory caused it some performance problems when running on multiple processors with consistent caches. For this reason, we tested another version of the matrix multiply kernel. The alternate version was produced by interchanging the "k" and "i" do loops in the program shown above. We chose this restructuring to keep the variable length do loop as the outermost one. This allows a parallelization strategy that creates n microtasks of a fixed granularity, the same technique as used in the previous version. With this strategy, each microtask computes the final result for one column of the PX array. This helps keep different microtasks from accessing values of the PX array that lie within the same cache line.

The indexing pattern of this second version produced unit strides for the PX and CX arrays, but not the VY array. This caused three additional cache misses for the entire multiply in the sequential case. Also, because the innermost loop in this case computes a final result for one element of the PX array, the intermediate results are accumulated in registers before being written to memory. As part of the execution of the "j" loop, some registers are used to hold values of the CX array, reducing the number of memory references in the inner loops. However, all available floating point registers are not used, and some values of the CX array are copied to the local stack instead. Although this version of the code produced only 3 additional cache

	ZS-1		Private		Shared	
Program	Time	Eff.	Time	Eff.	Time	Eff.
V. 1, n=25	2745	1.00	2745	1.00	2762	0.99
V. 1, n=50	5272	1.00	5272	1.00	5305	0.99
V. 2, n=25	3645	1.00	3645	1.00	3670	0.99
V. 2, n=50	7261	1.00	7261	1.00	7311	0.99

Table 1: Sequential code performance for different hardware configurations.

misses, it ran quite a bit slower than the previous version, averaging slightly less than 1 cycle per instruction. This slowdown is caused by the increase in instructions executed to copy the CX array as well as a reduction in execution overlap due to registers being busy during the multiplies and adds of the innermost loop.

Basic Performance

Table 1 shows the running time (in microseconds) and the efficiency of the first two sequential versions of the code for n=25 and n=50 on the different hardware configurations tested. The efficiency of sequential code on the ZS-1 is defined to be 1. The private column refers to the hardware cache consistency scheme where there is an index in the central directory that indicates which cache (if any) has a copy of a block. The shared column refers to the hardware consistency scheme where the directory contains an entry for each block that has a presence bit for each cache. The times for the ZS-1 and private cache consistency configurations are identical, because the cache consistency actions have no effect when only once processor and cache is in use. The times are slightly greater for the shared cache consistency configuration. This reflects the overhead necessary for a directory access on a write hit when the cache block is not already dirty. This situation occurs when data is read (and cached as non-dirty) before it is written. The extra overhead required in this case, however, is minimal.

Table 2 shows the performance for the sequential and parallel versions of the code running on the ZS-1 configuration. The original program is the version generated by the compiler for the uniprocessor. The chunk, self, and guided columns refer to the parallelized codes running on the uniprocessor without any instructions to implement software cache consistency. The running times for the parallel versions of the code are less than the original code in many cases. Although the parallel codes have additional instructions for scheduling and synchronization, their effect is more than offset by the elimination of instructions in the inner loops that is brought about by the parallelization. The eliminated instructions include calculation, comparison, and branch instructions for loop bounds as well as some "nop" instructions that were used to force the proper alignment. Certainly, the run-time support code added to parallelize the program does not significantly degrade performance when the code is run on a single processor.

Figures 1 and 2 show the speedups of the parallel programs for both versions of the code when n=50 and the codes are simulated for the unmodified ZS hardware. Figure 2 shows the speedups when calculated using the sequential times for both versions of the program. This is done to show both the speedup of this particular

	Original		Chunk		Self		Guided	
Program	Time	Eff.	Time	Eff.	Time	Eff.	Time	Eff.
V. 1, n=25	2745	1.00	2641	1.04	2630	1.04	2587	1.06
V. 1, n=50	5272	1.00	5216	1.01	5202	1.01	5108	1.03
V. 2, n=25	3645	1.00	3657	1.00	3720	0.98	3655	1.00
V. 2, n=50	7261	1.00	7274	1.00	7411	0.98	7272	1.00

Table 2: Sequential and parallel code performance the ZS-1.

code and to demonstrate the performance of the second version when it is compared to the "best sequential algorithm". The results are similar for both codes when n=25, but the speedups are about 15% less for 8, 12, and 16 processors. Also, for n=25, the three scheduling algorithms perform almost identically except in the case of 16 processors, where guided self-scheduling is about 11% slower and chunk scheduling is about 3% slower than self-scheduling.

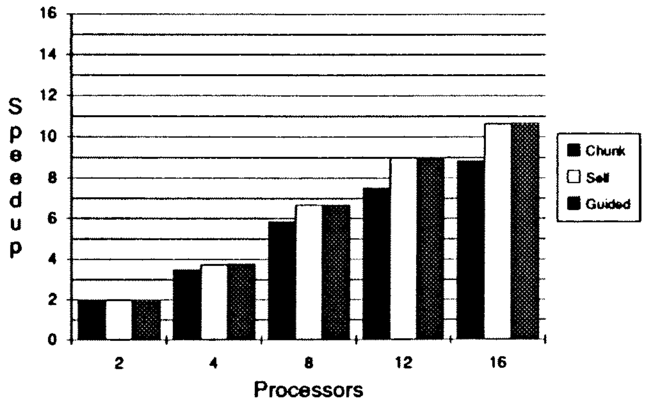

Figure 1: Speedups for n=50, first program version, assumed consistency.

The results in Figs. 1 and 2 represent an upper bound on parallel performance because software or hardware cache consistency has been omitted. The only slowdown in the code due to memory references are due to bank conflicts when multiple processors attempt to access main memory after a cache miss. Based on the small amount of scheduling overhead observed in the single processor results shown above and on the instruction counts for each processor, we can conclude that load balancing is the major limiting factor in the speedup of these programs.

Software Consistency

Perhaps one of the most interesting results was observed when cache consistency was enforced by making result data non-cacheable. Because the first version of the code does not place temporary results in registers, the speedups of the code in this case is never greater than 1. The second version of the code, however, writes to the result array much less frequently, and the speedups for n=50 when compared to both versions' sequential running times are shown in Fig. 3. These

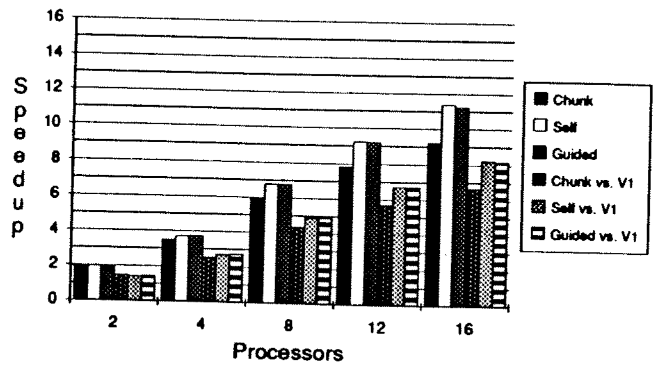

Figure 2: Speedups for n=50, second program version, assumed consistency.

results demonstrate that while non-cacheable pages may be a reasonable technique for maintaining consistency in some cases, such cases must be detectable at compile-time so that the appropriate restructuring can be done. If compile-time detection is not possible, poor performance will result.

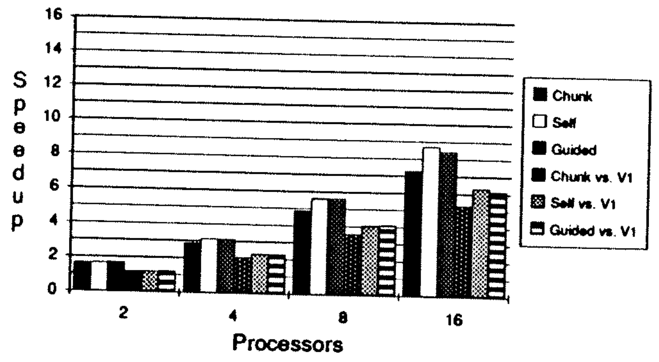

Figure 3: Speedups for n=50, second program version, non-cacheable result.

The performance of the codes implementing cache consistency with temporary result areas and cache management instructions was poor overall. For the first version of the program, the speedup never exceeded 1, and for the second version, it never exceeded 2.4. Although our algorithm for this consistency technique updated the final result area one processor at a time, we computed results based on the assumption that the merging of temporary result areas can be done pairwise in parallel using $\log_2 P$ steps where P is the number of processors. We computed the time needed for one reduction step by taking the differences between successive finishing times for all processors in our original version. We took an average of these differences (which were almost constant) and multiplied it by the number of steps required, $\log_2 P$. (We performed this calculation for values of P equal to 4, 8, and 16). This merging time was then added to the time taken by the processor that finished first in our simulations. This "base" time was only slightly greater than the times for assumed consistency shown above. This calculation of the merging time is optimistic, though, since is does not consider the extra synchronization required to implement this algorithm.

The calculated results for both versions of the source program using each of the scheduling algorithms were nearly identical, indicating that the software consistency actions dominated the running times. The speedup calculated versus the sequential running time of the first version of the source code was never greater than 2.5, while the speedup calculated versus the sequential running time of the second version of the source code was never greater than 3.5. The results did show an increase in speedup as more processors were employed, whereas the original code that updated the final result area one processor at a time showed its best performance at 2.4 with only 4 processors.

Hardware Consistency

For the hardware consistency tests, the simulator was modified as described in the EXPERIMENTAL TESTBED section above. Since cache consistency was enforced in hardware, there was no need for the programs to use any special cache management or data alignment instructions. The programs used for these tests were identical to those used in the assumed consistency tests shown in Figs. 1 and 2.

The performance of the private hardware scheme for cache consistency was similar to that of the calculated results for software consistency. For the first version of the program, the performance of hardware consistency was worse for fewer number of processors, and better for greater numbers of processors. For the second version of the source code, the private hardware consistency technique always performed worse than the software results, with maximum speedups being less than 2.7 and 2.0 when compared to both versions of the sequential code.

While there was not much variation depending on scheduling algorithms or the value of n for the second version of the source code, the first version showed much more variability. Figure 4 shows the results for the first version of the program with n=50. The results for n=25 show similar trends but smaller speedups, with the exception of chunk scheduling for 2 processors, which performs as well as guided self-scheduling.

It is interesting to note here the difference in performance of the various scheduling algorithms. In the other tests where consistency is assumed or main memory traffic is otherwise reduced, self-scheduling has a slight performance edge for larger numbers of processor because of its improved dynamic load balancing. However, in Fig. 4 we see that chunk scheduling and guided self-scheduling perform better in these experiments. Because they allocate chunks of iterations greater than 1, multiple adjacent iterations are allocated for each schedule. This provides additional spatial locality on each processor which reduces cache misses. This result is more visible for the first version of the program, where memory is accessed frequently.

The other interesting result in Fig. 4 is the performance of chunk scheduling for 16 processors. It is slower than the same program running on 12 processors, because of the chunk calculation algorithm. It computes an optimal chunk size of 1,

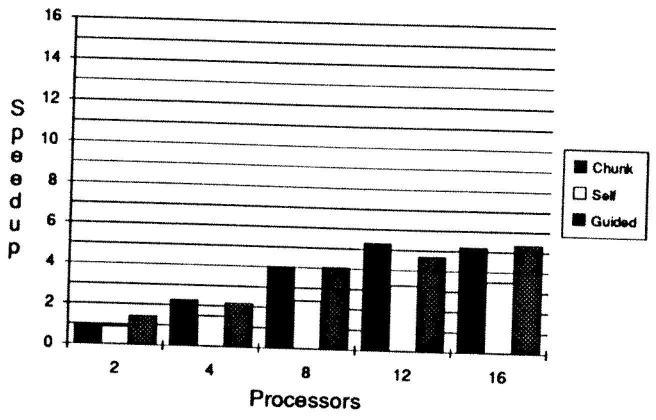

Figure 4: Speedups for n=50, first program version, private hardware consistency.

because that is the result of 25 **div** 16. However, for 12 processors, the chunk size is 2, with 1 extra chunk of size 1 required to finish the loop. The program requires 25 microtasks to run on 16 processors, and the scheduling introduces the locality problem observed with self scheduling. For 12 processors, only 13 microtasks are created, and the first 12 execute 2 adjacent iterations each.

The major problem with the private hardware consistency technique is the slowdown caused by the prevention of sharing of read-only data. This is especially evident for the second version of the source code, where each of the microtasks access the elements of one of the operands in the same order in a non-unit stride fashion. Since an access to any block must invalidate any other copies of that block, there are many more cache misses using this consistency technique.

The performance of the shared hardware consistency scheme was better than any of the other techniques, and approached the performance of assumed consistency for the second version of the source program. The shared hardware consistency mechanism allows any cache block to reside in more than one cache. As long as the block is not modified, no consistency actions are required. This approach avoids the difficulties encountered by the private scheme with read-only data.

Tables 3 and 4 show the running times for both versions of the program using shared hardware consistency. While the execution times for the first version of the source code are faster than those for private hardware consistency, they do show the same trends for the different scheduling algorithms. Because the second version of the program references memory much less often, the speedup of this code for for larger numbers of processors is greater. The performance of the second version of the program using 16 processors and shared hardware consistency is competitive with the first version of the program using assumed consistency. This version also shows more even performance between the different scheduling algorithms, again because memory is accessed less often. Figures 5 and 6 show the speedups for both versions of the program when n=50.

	Version 1 Times			Version 2 Times		
P	Chunk	Self	Guided	Chunk	Self	Guided
1	2656	2646	2603	3683	3745	3681
2	1420	3105	1387	1941	1977	1934
4	800	1697	909	1068	1080	1062
8	493	1032	628	639	634	627
12	394	744	637	498	490	489
16	740	742	632	377	364	405

Table 3: Parallel code running times for n=25 with shared hardware consistency.

	Version 1 Times			Version 2 Times		
P	Chunk	Self	Guided	Chunk	Self	Guided
1	5247	5235	5141	7324	7460	7322
2	2660	6193	2638	3686	3780	3716
4	1526	3294	1685	2090	1983	1964
8	910	1968	959	1219	1090	1082
12	705	1353	794	941	793	798
16	606	1120	732	812	653	651

Table 4: Parallel code running times for n=50 with shared hardware consistency.

Scheduling Algorithms

Our experiments did not suggest that any of the three scheduling algorithms tested was clearly the best. The results for assumed consistency suggest that self-scheduling and guided self-scheduling perform about the same, with chunk scheduling running slower. Self-scheduling provides the most speedup for 16 processors, while guided self-scheduling runs faster on fewer processors. Since self-scheduling allocates only 1 iteration for each schedule, it provides the best dynamic load balancing. However, guided self scheduling can come close in load balancing, since smaller chunks are allocated during later schedules. Also, since chunk scheduling and guided-self scheduling allocate more iterations per schedule on average, they have fewer scheduling points at run-time. For this reason, these approaches may incur less scheduling overhead than self-scheduling, even though their scheduling code requires more instructions to compute chunk size and loop bounds.

For the other tests, the performance of the scheduling algorithms depended on the amount of memory accesses in the code. For the first version of the source program, where the number of memory accesses is great, the performance of chunk scheduling was best, with guided self-scheduling being next. As mentioned earlier, this is due to the spatial locality in memory referencing that occurs when adjacent iterations are executed on the same processor. For the second version of the program, where memory accesses are much fewer in number, the pattern described above for assumed consistency was observed, with self-scheduling providing the most speedup in most cases.

These results suggest that different scheduling techniques be used for depending

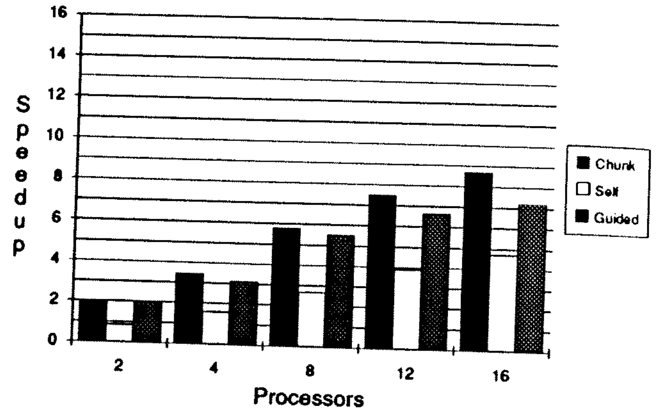

Figure 5: Speedups for n=50, first program version, shared hardware consistency.

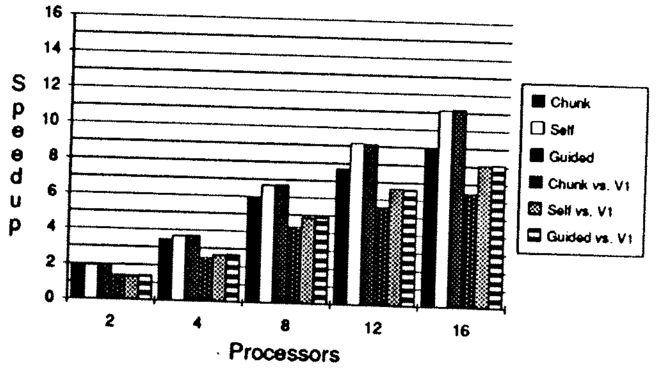

Figure 6: Speedups for n=50, second program version, shared hardware consistency.

on the characteristics of the code inside the parallel loop. However, having seen that the code that accesses memory the least is the most likely to achieve the highest speedup, it seems that compilers should optimize in favor of this type of code, and use self-scheduling because it is easiest. More experiments using larger grain sizes and different source codes are needed before any firm conclusions can be made regarding the scheduling algorithm used.

CONCLUSIONS AND FUTURE WORK

While our investigation studied several aspects of the performance of parallel loops on a non-bus multiprocessor, it should not be assumed that the results observed in our tests will be repeated on other multiprocessors or with different parallel programs. This work is preliminary and much remains to be done before definite conclusions can be stated. We can, however, reach the following preliminary conclusions based on our experiments.

Our results show shared hardware consistency to be the best technique for maintaining cache consistency. It outperformed the other techniques we tested, including private hardware consistency. This differs with the conclusions offered in [4], where it is conjectured that the performance of shared and private hardware consistency is roughly equal. This may be attributable to the different types of parallelism present in the test programs. We found that a fine-grained parallel program is more likely to shared data than those parallelized at the program level. Also, our tests accounted for the effects of sharing cache blocks even when individual words are not shared. This level of sharing caused too many invalidations in the private hardware consistency case.

The software schemes we tested did not fare well at all, with the possible exception of non-cacheable data for the second version of the source program. The overhead of temporary result areas for consistency seems to be too great for microtasking. Experiments using programs with much higher granularities should be run. However, as the size of the result area grows, so does the overhead in merging these areas. As for non-cacheable data, the performance is directly dependent on the number of references to this data. In some cases, the effect can be devastating. If this technique were to be used seriously in a real machine, hardware techniques to speedup these accesses without blocking the cache should be employed.

While the shared hardware consistency approach performed the best in our tests, there are other tradeoffs to consider when choosing a cache consistency technique. The major disadvantage of this hardware technique is the size of the central directory required. Since a presence bit is required for each cache in the system, the size of the directory is proportional to the number of caches in the system as well as the size of main memory. This situation can prevent the use of this technique in systems with a large number of processors.

There are also problems and tradeoffs to consider when using software schemes for consistency. While the techniques we tested did not perform well, it is possible that techniques using cache management instructions embedded within the parallel loop bodies may perform better. However, these techniques require sophisticated compilers, as well as protection from interrupts or context switching by the operating system. The compiler has to manage the mapping between memory words and cache blocks, to ensure that implicit block sharing does not introduce inconsistency. If the software also assumes the presence of, or absence of, cache blocks based on prefetching or cache management instructions, the operating system must prevent context switching and microtask migration from violating these assumptions. For these reasons, we believe that software consistency schemes are best suited for single user compute-intensive parallel processing systems.

Finally, we can state some basic conclusions about our experience with parallel processing at the microtasking level. First, different optimization strategies must be employed by the compiler depending on the number of processors targeted and the cache consistency technique used. As we have seen in the results listed above, a version of a program that runs slower on a single processor may run faster on multiple processors. Also, optimizing for fewer memory references reduces cache consistency overhead and allows a different scheduling algorithm to be used.

Run-time support overhead for microtasking is not significant, but load balancing and overhead for cache consistency are problems. The single processor results demonstrated that run-time support overhead for parallelism was not expensive. This is probably due, at least in part, to the low overhead synchronization provided by the semaphore registers. The results for multiple processors and assumed consistency demonstrated that load balancing was a limiting factor in speedup for larger numbers of processors. Also, we saw from the results of tests that included cache consistency techniques that the performance of assumed consistency could never be matched. With the exception of shared hardware consistency, this overhead was a major factor in the slowdown of the parallel program.

More parallelism of varying grains is needed. This conclusion follows directly from the previous point. Mechanisms to create more microtasks that are ready to run at any given time should help the load balancing problem. Although this may require more overhead in run-time support code, the tradeoff may well be worth it, since this overhead is quite low with the current technique. Our results also show a maximum efficiency less than 75% with 16 processors. If the source program has a significant fraction of sequential code between parallel loops, overall program efficiency will be quite low. Language extensions or alternate loop restructuring techniques are needed to exploit more parallelism and raise this efficiency level.

In order to confirm all of our conclusions and provide more detailed results, more experiments need to be conducted. We are currently studying the speedup potential of different doacross loops, where cross iteration data dependencies exist that require additional synchronization. We are also developing some language extensions and their run-time support code to provide more parallelism of varying grain sizes. In addition to these new approaches, we also plan to test more loops and complete programs to study the speedup potential of parallel loops with very large iteration counts as well as the effects of sequential sections in complete programs.

Acknowledgments This work is supported in part by the Astronautics Corporation of America and by NSF grant MIP-8802771.

References

[1] J. Archibald and J. L. Baer, "Cache coherence protocols: Evaluation using a multiprocessor simulation model," *ACM Trans. Computer Systems*, vol. 4, no. 4, pp. 273–298, Nov. 1986.

[2] S. J. Eggers and R. H. Katz, "The effect of sharing on the cache and bus performance of parallel programs," in *Proc. 3rd Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pp. 257–270, Apr. 1989.

[3] S. J. Eggers and R. H. Katz, "Evaluating the performance of four snooping cache coherency protocols," in *Proc. 16th Int. Symp. Computer Architecture*, pp. 2–15, June 1989.

[4] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An evaluation of directory schemes for cache coherence," in *Proc. 15th Int. Symp. Computer Architecture*, pp. 280–289, June 1988.

[5] S. Owicki and A. Agarwal, "Evaluating the performance of software cache coherence," in *Proc. 3rd Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pp. 230–242, Apr. 1989.

[6] D. Kuck, Y. Muraoka, and S. Chen, "On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speedup," *IEEE Trans. Computers*, vol. C-21, no. 12, , Dec. 1972.

[7] R. Allen and K. Kennedy, "PFC: A program to convert FORTRAN to parallel form," Tech. Rep. MASC-TR 82-6, Dept. of Mathematical Sciences, Rice University, March 1982.

[8] M. D. Guzzi, D. A. Padua, J. P. Hoeflinger, and D. H. Lawrie, "Cedar FORTRAN and other vector and parallel FORTRAN dialects," in *Proc. Supercomputing '88*, pp. 114–121, Nov. 1988.

[9] B. Leasure et al., "PCF FORTRAN: Language definition," Tech. Rep. Version 1, The Parallel Computing Forum, Aug. 1988.

[10] A. J. Musciano and T. L. Sterling, "Efficient dynamic scheduling of medium-grained tasks for general purpose parallel processing," in *Proc. 1988 Int. Conf. Parallel Processing*, pp. 166–175, Aug. 1988.

[11] A. Norton and W. L. Chang, "Self-scheduling in the runtime environment," Technical Report RC 12572 (#56256), IBM T. J. Watson Research Center, Yorktown Heights, NY, Feb. 1987.

[12] M. Wolfe, "Multiprocessor synchronization for concurrent loops," *IEEE Software*, pp. 34–42, Jan. 1988.

[13] C. D. Polychronopoulos, *Parallel Programming and Compilers*, Kluwer Academic Publishers, Norwell, MA, 1988.

[14] B. J. Smith, "Architecture and applications of the HEP multiprocessor computer system," in *Real Time Processing IV, Proc. of SPIE*, pp. 241–248, 1981.

[15] C. D. Polychronopoulos and D. J. Kuck, "Guided self-scheduling: A practical scheduling scheme for parallel supercomputers," *IEEE Trans. Computers*, vol. C-36, no. 12, pp. 1425–1439, Dec. 1987.

[16] H. Cheong and A. V. Veidenbaum, "A cache coherence scheme with fast selective invalidation," in *Proc. 15th Int. Symp. Computer Architecture*, pp. 299–307, June 1988.

[17] R. Cryton, S. Karlovsky, and K. P. McAuliffe, "Automatic management of programmable caches," in *Proc. 1986 Int. Conf. Parallel Processing*, pp. 229–238, Aug. 1986.

[18] M. Beltrameti, K. Bobey, and J. R. Zorbas, "The control mechanism for the myrias parallel computer system," *ACM Computer Architecture News*, Aug. 1988.

[19] L. M. Censier and P. Feautrier, "A new solution to coherence," *IEEE Trans. Computers*, vol. C-27, no. 12, pp. 1112–1118, Dec. 1978.

[20] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, New York, 1984.

[21] A. Gottlieb et al., "The NYU Ultracomputer—Designing a MIMD, shared memory parallel machine," *IEEE Trans. Computers*, vol. C-32, pp. 175–189, Feb. 1983.

[22] M. Dubois, C. Scheurich, and F. A. Briggs, "Synchronization, coherence, and event ordering in multiprocessors," *IEEE Computer*, pp. 9–21, Feb. 1988.

[23] Cray Research Inc., *Cray X-MP Mainframe Reference Manual*, Minneapolis, MN, 1982.

[24] Astronautics Corporation of America, *ZS Central Processor – Architecture Reference Manual*, Madison, WI, 1988.

[25] B. Beck, B. Kasten, and S. Thakkar, "VLSI assist for a multiprocessor," in *Proc. 2nd Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pp. 10–20, Oct. 1987.

[26] J. E. Smith, "Dynamic instruction scheduling and the Astronautics ZS-1," *IEEE Computer*, vol. 22, no. 7, pp. 21–35, July 1989.

[27] J. E. Smith, "Decoupled access/execute computer architectures," *ACM Trans. Computer Systems*, vol. 2, no. 4, pp. 289–308, Nov. 1984.

[28] R. L. Lee, P.-C. Yew, and D. H. Lawrie, "Data prefetching in shared memory multiprocessors," in *Proc. 1987 Int. Conf. Parallel Processing*, pp. 28–31, Aug. 1987.

[29] C. K. Tang, "Cache system design in the tightly coupled multiprocessor system," in *AFIPS Proc. National Computer Conf.*, vol. 45, pp. 749–753, 1976.

[30] F. H. McMahon, "The livermore FORTRAN kernels: A computer test of the numerical performance range," Tech. Rep. UCRL-53745, Lawrence Livermore National Laboratory, Livermore, Calif. 94550, Dec. 1986.

Predicting the Performance of Shared Multiprocessor Caches

Hendrik A. Goosen and David R. Cheriton
Computer Science Department
Stanford University

Abstract

We investigate the performance of shared caches in a shared-memory multiprocessor executing parallel programs, and formulate simple models for estimating the load placed on the bus by such a shared cache. We analyze three parallel program traces to quantify the amount of sharing that takes place during program execution. These results indicate that shared caches can substantially reduce the load placed on a bus by a large number of processors.

Keywords: shared-memory multiprocessors, shared cache, data reference characteristics.

1 INTRODUCTION

There is considerable interest in the design of scalable shared memory multiprocessors. The problem of building such machines is largely that of building a memory system that is fast enough to supply the multiple processors with the data they need to execute programs and communicate with each other.

Modern microprocessors require a multi-level cache design to approach peak performance [8], and some processors already have large instruction and data caches on-chip (e.g., 12 Kbytes in the Intel i860). In such a multilevel cache hierarchy, the majority of the traffic (90% to 99% in uniprocessors with large cache blocks [9]) is absorbed by the first-level cache, which means that the higher level caches are idle most of the time.

In a multiprocessor system, the utilization of a higher-level cache can be increased by sharing it among several processors. Sharing a cache between several processors executing the same parallel program can also