A Performance Evaluation of Cache Injection in Bus-based Shared Memory Multiprocessors

Aleksandar Milenkovic, Veljko Milutinovic*

University of Alabama in Huntsville, *University of Belgrade
E-mail: milenka@ece.uah.edu, vm@etf.bg.ac.yu

Abstract

Bus-based shared memory multiprocessors with private caches and snooping write-invalidate cache coherence protocols are dominant form of small- to medium-scale parallel machines today. In these systems the high memory latency poses the major hurdle in achieving high performance. One way to cope with this problem is to use various techniques for tolerating high memory latency. Software-controlled cache prefetching and data forwarding are two widely used techniques for tolerating high memory latency in scalable cache-coherent shared memory multiprocessors. However, some previous studies have shown that these techniques are not so effective in bus-based shared memory multiprocessors. In this paper, we propose a novel software-controlled technique called cache injection, which combines consumer and producer initiated approach, and broadcasting nature of bus. Performance evaluation based on program-driven simulation and a set of scientific applications and test benchmarks shows that cache injection is highly effective in reducing misses and bus traffic.

1. Introduction

The popularity of bus-based shared memory multiprocessors or symmetric multiprocessors (SMPs) has greatly increased since almost all modern microprocessors include the support for building cost-effective bus-based SMPs [2]. In bus-based SMPs private caches and write-invalidate protocols are essential to reduce bus congestion and to maintain data coherence, respectively. However, the widening speed gap between processor and memory, high contention on the bus, and data sharing in parallel programs cause the two performance bottlenecks: large latencies associated with read and write cache misses, and bus traffic. While the latency of write misses can be successfully hidden by appropriate write-buffers and relaxed memory consistency models [12], the latency of read misses still remains. Read misses can be classified into cold, coherence, and replacement. Cold miss occurs if the requested block has never been referenced by the processor, coherence miss occurs if the block has been referenced by the processor, but has been written to by another processor, while all other misses are referred as replacement misses, since they are caused by replacements from the cache due to its limited size and associativity. To cope with this problem, researchers have proposed techniques that reduce the number of read misses and bus traffic, such as software-controlled cache prefetching and data forwarding.

In order to illustrate how each technique works we will use a simple example, which demonstrates producer-consumer sharing pattern where processor P0 produces, and processors P1 and P2 consume the data (Figure 1a). In software-controlled cache prefetching, a processor executes a special `Pf` instruction, which initiates a non-blocking fetch operation that brings a data block, expected to be used by that processor, into its cache [10]. Ideally, the data block arrives at the cache before it is needed by the processor, and its load instruction results in a cache hit (Figure 1b, processor P2). However, cache prefetching is useful even if it is not issued early enough; in that case the latency will be only partially hidden (Figure 1b, processor P1). For many programs and sharing patterns (e.g., producer-consumer), producer-initiated data transfers are a natural style of communication. Producer initiated primitives are known as data forwarding, delivery, remote writes, and software-controlled updates [5]. With data forwarding, when a processor produces the data (Fig 1c, processor P0), in addition to updating its cache, it sends a copy of the data to the caches of the processors that are identified by compiler or programmer as its future consumers. Therefore, when the consumer processors access the data block, they find it in their caches (Figure 1c, processors P1 and P2). To support this, a special `StoreForw` instruction is needed; this instruction performs an ordinary store operation, and then initiates transactions to deposit new value in the caches of the specified processors.

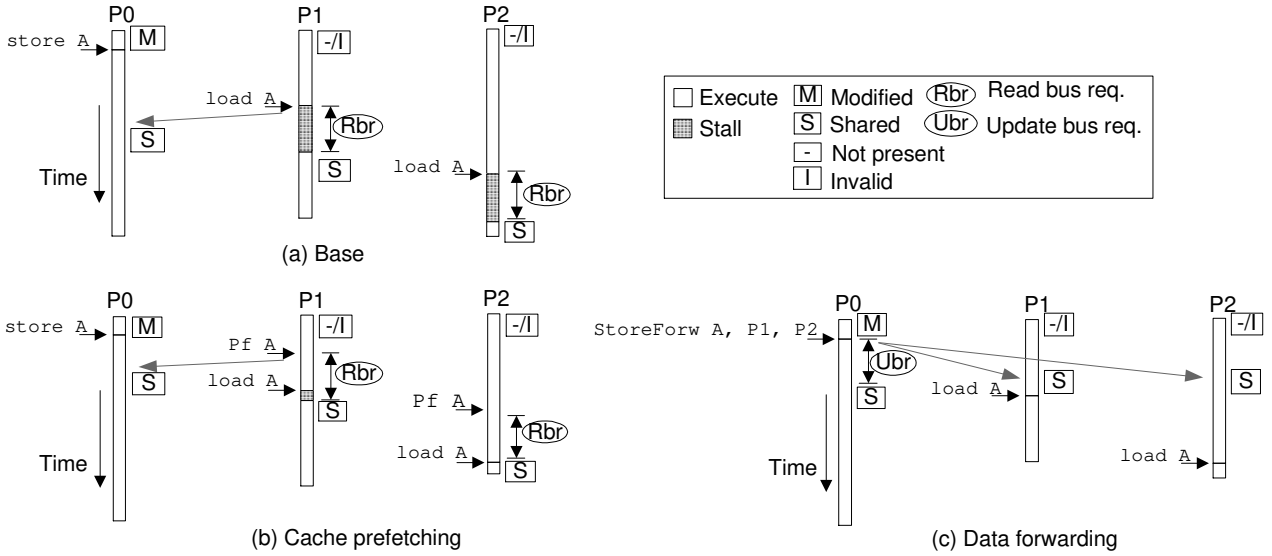

Figure 1. Cache prefetching and data forwarding.

Comparing cache prefetching and data forwarding, it should be noted that prefetching can eliminate any kind of read misses (cold, conflict, and coherence), while forwarding can eliminate coherence and in some cases cold misses. However, prefetching is inapplicable or insufficient when the location to be accessed is not known sufficiently early or when the value to be read is not produced sufficiently early (e.g., for synchronization variables and producer-consumer sharing patterns). In addition to that, prefetching can negatively affect data sharing when it is initiated too early, before the producing processor finishes the data producing; in such cases cache prefetching can degrade performance. For the coherence misses, forwarding can be more effective than prefetching since forwarding delivers a data block to consumers as soon as it is produced and since a single instruction is enough to initiate forwarding to several consumers. However, data forwarding requires more sophisticated compiler support, compared to prefetching; for prefetching, consumer processor does not need to know the identity of the producer processor, while for forwarding the producer processor needs to know the identity of consumers.

Most of the studies [1, 5, 10, 13-16] examined the effectiveness of cache prefetching and data forwarding in CC-NUMA or CC-UMA architectures, except [18, 19], which examined the potential of cache prefetching in bus-based multiprocessors. This study reported poor effectiveness of cache prefetching, despite assumed high memory latency. The main reasons for that are the following. First, prefetching increases bus traffic. Since bus-based architecture is very sensitive to changes in bus traffic, prefetching can result in performance degradation. Second, cache prefetching can negatively affect data sharing, especially in the cases of prefetching initiated too early. Last, current prefetching algorithms are not so effective in predicting coherence misses. Actually, cache misses caused by data sharing represent the biggest challenge for designers, especially as caches become larger and coherence misses dominate the performance of parallel programs. On the other side, data forwarding has not been examined in bus-based architectures yet. Complexity of implementation and compiler algorithm restricts applicability of data forwarding in bus-based architectures. Dahlgren et al. explored the effectiveness of the software-controlled update in bus-based multiprocessors, where a special instruction initiates update of all invalid copies of the specified cache block in the system [3, 4]. This approach requires less sophisticated compiler support since it does not require identification of future consumers and it can be implemented at low cost. However, this approach is less flexible than classic data forwarding as defined in [5], because it does not allow forwarding to the processors not having the invalid copies of the data block.

In this paper a novel technique called cache injection is proposed. Using advantages of existing techniques, cache prefetching and data forwarding, and the characteristics of bus-based architectures, cache injection overcomes some of the shortcomings of the existing techniques, such as high contention on the bus, negative impact on data sharing and instruction overhead in the case of cache prefetching, compiler complexity in identification of future consumers, and complexity of implementation in the case of data forwarding. Cache injection, based on snooping possibilities inherent to bus-based architectures, is aimed to reduce coherence cache misses and bus traffic. In cache injection, data consumers initialize local injection tables with addresses of data expected to be used. Those data are injected into consumer caches during a read bus transaction, initiated by some other consumer, or during a software-initiated write back bus transaction,

initiated by a data producer. The proposed technique can be combined with the existing ones in order to raise the overall effectiveness of techniques for tolerating memory latency in bus-based multiprocessors.

We have evaluated the performance of cache injection on two synchronization kernels, three test benchmarks, and four parallel applications. In the case of synchronization kernels, cache injection shows gains from 12% to 96% over the base system. In the case of test benchmarks and applications, cache injection improves performance from 2% to 90% relative to the base system, depending on the number of processors and parameters of the memory subsystem.

In the following section, we define cache injection and discuss its programming model and its implementation in a bus-based shared memory multiprocessor. Section 3 describes experimental methodology used in performance evaluation. Section 4 presents results of the experiments. Section 5 discusses related work. Section 6 summarizes current and discusses possible future work.

2. Cache Injection

In cache injection, a consumer predicts its future needs for shared data by executing an `OpenWin` instruction. This instruction does not initiate any bus transaction, but only stores the first and the last address of a range of consecutive cache blocks – an address window - in a special local injection table. There are two main scenarios when cache injection can happen: during a bus read transaction (injection on first read) or during a software-initiated write-back bus transaction (injection on write-back).

Injection on first read is applicable when there is more than one consumer. Each consumer initializes its injection table according to its future needs. When the first one among consumers executes a load instruction specifying the shared data, it sees a cache miss and initiates a bus read transaction. During this transaction, each cache controller snoops the bus and if there is an injection hit - the address of the currently transferred block belongs to one of the open address windows - the processor stores the block into its cache. Hence, in the case of multiple consumers, only one read bus transaction is needed to update all consumers, if they all have initialized their injection tables before this transaction. In our example both consumer processors, P1 and P2, initialize their injection tables using `OpenWin` instructions. Processor's P1 `load` instruction sees a cache miss, and, consequently, initiates read bus transaction. During this transaction, processor P2 injects that block into its cache; hence, its `load` will see a hit in the cache (Figure 2a).

Injection on write-back bus transaction is applicable when shared data exhibit 1-Producer-1-Consumer, 1-Producer-Multiple-Consumers, or migratory sharing patterns. Here each consumer also initializes its injection table. After data producing is finished, the producer initiates a bus write-back transaction in order to update the memory, by executing a `StoreUp` instruction. During the write-back bus transaction, each consumer snoops the bus, and if it finds an injection hit, it stores the data block into its cache. Injection on write-back is shown on Figure 2b. In this scenario, processor P0 replaces the last `store` with a `StoreUp` instruction, which initiates write-back bus transaction. During this transaction, processors P1 and P2 inject that data block into their caches.

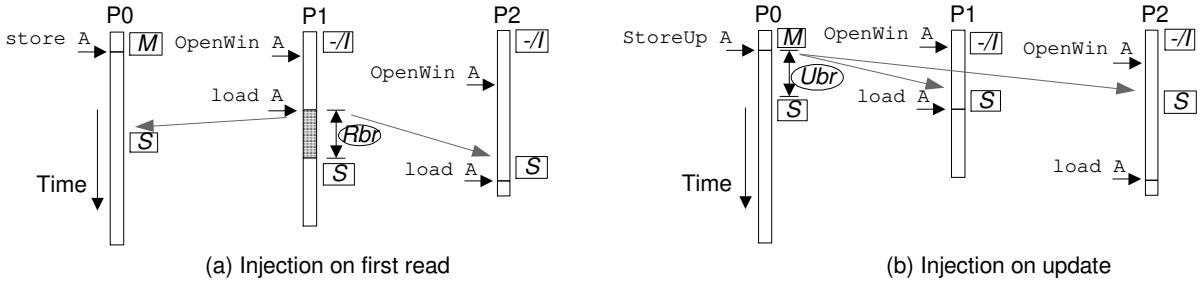

Figure 2. Illustration of injection mechanism.

Hardware support for cache injection includes injection table, proposed instructions, and a negligible modification of bus control unit. Injection table is implemented as a part of the cache controller. Each entry includes two address fields, *Laddr* and *Haddr*, which define the first and the last address of an address window, respectively, and a valid bit *V*. We use the random replacement policy. The organization of injection table is shown in Figure 3. Proposed instructions are:

- `OpenWin <Laddr><Haddr>`
  Initializes an entry in the injection table, by setting the valid bit and putting *Laddr* and *Haddr* values in the corresponding entry fields. If only one cache block should be injected, *Laddr=Haddr*.
- `CloseWin <Laddr><Haddr>`

Checks injection table, and if there is an open window with specified *Laddr* and *Haddr*, it closes that window by resetting the valid bit.

- `Update <Addr>`
  Checks the cache and if the specified cache block is modified, it initiates the write-back bus transaction and changes the block state into *Shared*; otherwise, it acts like `noop` instruction [16].
- `StoreUp <Addr><Value>`
  Performs an ordinary `Store` instruction; in addition, it initiates write-back bus transaction [16].

Compiler and/or programmer insert `OpenWin` and `CloseWin` instructions at the consumer side, and `Update` and `StoreUpdate` instructions at the producer side, in the injection on write-back scenario. Compiler can reduce instruction overhead by replacing an ordinary `store` instruction, followed by an `Update`, with a `StoreUp` instruction. The injection is performed at the cache block level and the new state of injected cache block is always *Shared*. In order to support exclusive transfer, `UpdateInv` and `StoreUpInv` instructions could be used. By using these instructions, a producer invalidates cache block in its cache. In this paper, we have not considered the use of these instructions.

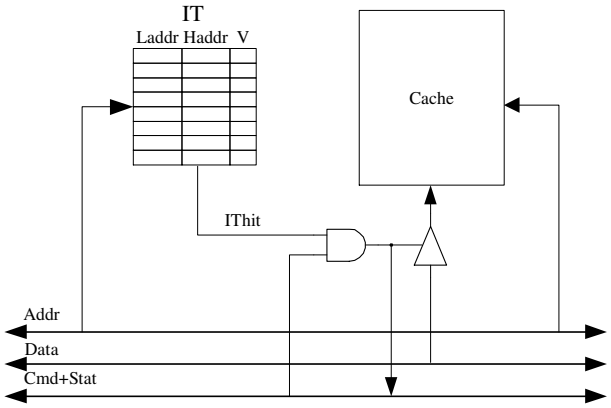

Figure 3. Organization of the injection table.

3. Experimental Methodology

We evaluate performance impact of cache injection using Limes [7] – a tool for program-driven simulation of shared memory multiprocessors. As a simulation workload we use two synchronization kernels (LTEST and BTEST), three parallel test applications (PC, MM, and Jacobi) well suited to demonstrate various data sharing patterns such as 1-Producer—Multiple-Consumers, read only with multiple consumers, and 1-Producer—1-Consumer, and four parallel applications from SPLASH-2 suite (Radix, FFT, LU, and Ocean) [20]. Proposed instructions for cache injection support are hand-inserted. A detailed simulator of memory subsystem is developed. For each application, we compare the performance of base system and one or more systems that include cache injection. Experiments vary different memory subsystem parameters, in order to explore their influence on cache injection.

The modeled architecture is a bus based shared memory multiprocessor system with the MESI write-back invalidate cache coherence protocol. The bus supports split transactions and uses round robin arbitration scheme. We assume a single-issue, in-order processor model with blocking reads. Processors execute a single cycle per instruction. Each processor includes only a first level cache memory. We assume that instructions always hit into the cache. Cache hit is solved without penalty. Figure 4 shows the structure of the modeled cache controller and Table 1 specifies relevant system parameters. The read and the read-exclusive bus transactions include the request and the response phases. The memory read cycle (MRC) defines time needed to retrieve a requested block from memory. A two-word transfer via the data bus takes 2pclk (pclk - processor cycle); hence, the block transfer takes 8pclk. It is assumed that the memory controller buffer has enough capacity to accept each block during write-back bus transactions at the data bus speed (MWC=1pclk). Lock sleep counter defines time between two successive lock acquires when the lock is busy. We assume 128-entry injection tables, although experiments show that most of the applications need less IT entries.

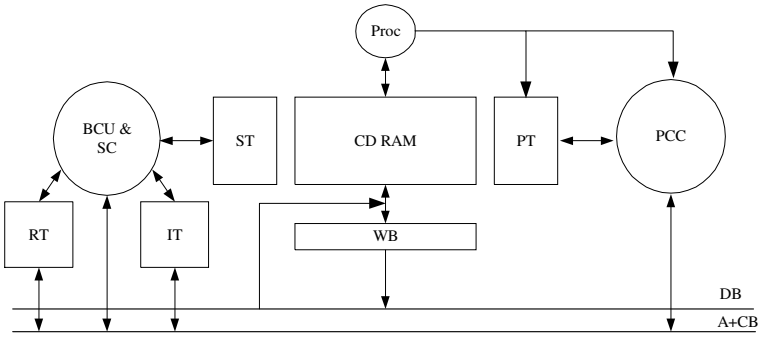

Figure 4. Cache Controller Structure.
Proc – Processor, BCU&SC – Bus Control Unit & Snoop Controller, PCC – Processor Cache Controller, ST – Snoop Tags, PT – Processor Tags, CD – Cache Data RAM, IT – Injection Table, RT – Request Table, WB –Write-Back Buffer, DB – Data Bus, A+CB – Address + Control Bus.

Table 1. Simulation parameters.

Parameter	Value
Cache size	32/64/128KB, 1024KB
Cache line size	32B (8W)
Data bus width	8B (2W)
Memory read cycle (MRC)	20, 100 pclk
Memory write cycle (MWC)	1 pclk
Snoop cycle	2 pclk
Lock sleep counter	5 pclk
IT size	128
WB size	32B

The aim of our evaluation is to determine the upper bound of performance benefit of cache injection before we start developing compiler support. Hence, we use simple heuristics based on application behavior to insert instructions for cache injection by hand. Here, we will discuss our heuristics, separately for synchronization variables and for true shared data.

Support for injection of synchronization variables is accomplished using injection on first read, since this approach does not require any modification of synchronization operations. This support is quite simple and includes the initialization of the injection table before a synchronization event and the invalidation of the corresponding entry in the injection table after the synchronization is finished. Thus, before a `lock(L)` operation, we insert an `OpenWin(L,L)` instruction, and after an `unlock(L)` operation, we insert a `CloseWin(L,L)` instruction. Similar procedure is used for global synchronization primitive `Barrier(B,N)`. It is clear that inserting instructions to support injection of synchronization variables can be solved by using macros that expand synchronization operations. Hence, the true challenge is the compiler support for injection of true shared data.

If there is an 1-Producer—Multiple-Consumers sharing pattern, injection on first read or injection on write-back can be used. Although injection on write-back may be more efficient, we use injection on first read because it implies no action at the producer side. However, if sharing pattern is 1-Producer—1-Consumer, we have to use injection on write-back. In this case, there should be supported not only the initialization of injection tables of consumers, but also the insertion of `Update` instructions.

4. Results

This section is organized as follows. First subsection gives a short overview of simulation results. Second subsection discusses the influence of the injection mechanism on synchronization kernels. Third subsection explains the data sharing patterns and injection support and discusses results for applications PC, MM, Jacobi, Radix, LU, FFT, and Ocean.

4.1. Overview

For synchronization kernels LTEST and BTEST we compare average lock acquire time (LAT) and execution time (ET) for the base system (*B*) and the system with injection (*I*), depending on the number of processors and memory read cycle time. Figure 5 shows LAT and normalized execution time (NET) for LTEST and BTEST, when MRC=20pclk and MRC=100pclk.

For applications PC, MM, and Jacobi we compare execution time for the base system (*B*) and the system which includes support for injection of synchronization variables and true shared data (*Isd*). In order to examine the sensitivity of cache injection on number of processors, cache size, and MRC, experiments are repeated with various number of processors (1, 2, 4, 8, 16, and 32), cache sizes of 64/128KB and 1024KB, and MRC values of 20pclk and 100pclk. Figure 6 shows speedup for the evaluated systems *B* (SU_B) and *Isd* (SU_I). Speedups are calculated using the following formula, where $ET_B(P)$ represents the execution time on the base system with P processors.

$$SU_B(P) = ET_B(1) / ET_B(P),\ SU_I(P) = ET_B(1) / ET_B(P)$$

For applications Radix, LU, FFT, and Ocean we compare execution time for the base system (*B*), the system which supports injection of synchronization variables only (*Is*), and the system which supports injection of both synchronization variables and true shared data (*Isd*). Figure 7 illustrates speedup for the evaluated systems with different parameters of the memory subsystem.

According to the experiment results, the injection mechanism almost always improves performance. In systems with relatively small number of processors (2 and 4), and small cache size, for applications Jacobi, FFT, and Ocean, the cache injection sometimes may degrade performance. However, in those cases the degradation percentage is negligible (never over 2%). According to the expectations, the efficiency of the injection increases with increase of the number of processors in the system, cache memory size, and memory read cycle time. When the number of processors increases, the percentage of shared data increases, as well as the number of sharers, hence the benefit of injection increases due to lowering miss rate and reducing bus traffic. Larger cache memory size reduces probability of collision of the injected data and the current working set. If the memory read cycle time is longer, there is more to gain by reducing the read stall time. One exception of above trends is the application MM, where improvement is less in systems with larger cache size and longer memory cycle time. This anomaly will be explained in the section dedicated to this application. Solutions *Isd* have better performance than solutions *Is* for all experiments with Radix, and in majority of experiments with FFT. For LU and Ocean solutions *Is* are predominantly better than *Isd*. The reasons for this phenomenon will be explained in the following sections.

4.2. Synchronization kernels

The influence of cache injection on locks is evaluated using LTEST synchronization kernel. This kernel simulates 1000 requests per processor to enter the critical section; duration of the critical section is 200pclk, and delay between release of the lock and next attempt to acquire it is defined using uniform distribution 0-1000 pclk. Locks are implemented using the test&test-and-set algorithm.

For the LTEST, the system with injection assumes the support for injection of lock variable, which protects the critical section. The instruction `OpenWin` is inserted at the beginning of the kernel. When one processor initiates the read cycle, reading the lock variable from memory during testing phase, all other processors will inject the lock variable (the injection on first read). This mechanism provides significant bus traffic reduction, especially in the conditions of high lock contention. The alternative approach is the injection on write-back. This approach requires the modification of synchronization primitives: the modified `lock` initiates write-back bus cycle after a processor sets a lock variable to busy and the modified `unlock` initiates the write-back bus cycle after the processor resets it to free. However, simulation results show a negligible improvement over the solution described above; therefore, we concentrate on the injection on first read.

Figure 5a and Figure 5b show lock acquire time for base system and system with injection, for MRC=20pclk and MRC=100pclk, respectively, depending on the number of processors (P). The cache injection reduces LAT between 27% (P=4) and 75% (P=32) when MRC=20pclk, and between 66% (P=4) and 77% (P=32) when MRC=100pclk. Figure 5c and Figure 5d show normalized execution time for LTEST. For LTEST, cache injection reduces ET between 12% on a system with 4 processors and 79% on a system with 32 processors, when MRC=20pclk. When MRC=100pclk, ET is reduced between 48% and 84%.

The influence of cache injection on barriers is evaluated using BTEST synchronization kernel. In this kernel, each processor performs global synchronization 100 times, and there are 120pclk between two subsequent barriers. For the

BTEST, the system with injection assumes the support for injection of two lock variables and a counter used in an implementation of a barrier primitive. Since the implementation is based on lock and unlock primitives, the benefit is the consequence of the same mechanism as described above. Figure 5e and Figure 5f show NET for BTEST. For BTEST, cache injection reduces ET between 56% on a system with 4 processors and 94% on a system with 32 processors, when MRC=20pclk. When MRC=100pclk, ET is reduced between 63% and 96%. Higher improvements for this kernel, compared to LTEST, are a consequence of higher contention.

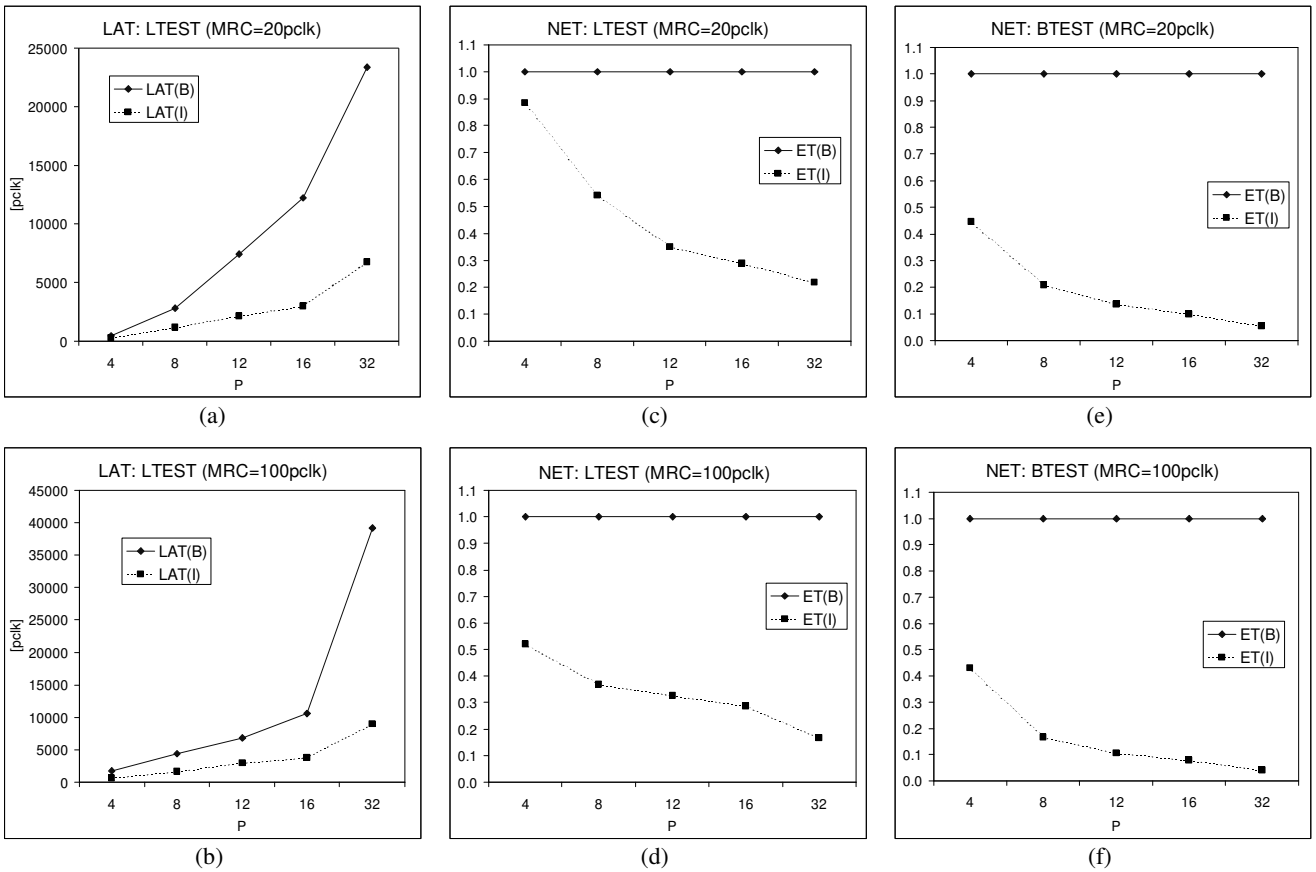

Figure 5. LAT and NET: LTEST, BTEST.

4.3. Applications

PC. PC iterates *i* times over two-phase loop. In the first phase, each processor computes a parameter value depending on its ID and all elements of shared matrix. In the second phase, each processor modifies the elements of the submatrix assigned to it, using previously computed parameter. Hence, there is an 1-Producer-(P-1)-Consumers sharing pattern between successive iterations (P is the number of processors). The matrix size is 128×128, *i*=20. The coherence misses dominate since each processor modifies its assigned submatrix, which is read by all other processors in the next iteration. The support for the injection is achieved by insertion of only one instruction, which defines an address window encompassing the whole shared matrix.

Here it should be noted that the absolute value of SU is not of interest, because the problem dimension grows with the increase of the number of processors – during the first phase each processor reads all elements of the shared matrix. The real indicator of the injection benefit is the relative improvement of execution time of *Isd* system over *B* system. The cache injection improves performance from 2.1% in the system with P=2, to 49.4% in the system with P=32, depending on parameters of the memory subsystem (Figure 6a, b).

MM. MM is a parallel version of matrix multiplication A=AxB. Each processor computes elements of the assigned submatrix of matrix A, so all processors read matrix B elements (read only data with P consumers). The matrix size is

128×128. As all processors read elements of the shared matrix B, the injection on first read significantly reduces bus traffic. To support the injection, each processor defines an address window encompassing the whole matrix B.

It is interesting to notice that for this application the efficiency of cache injection decreases as the cache memory size increase (Figure 6c, d). This is a consequence of multiple injections in a system with small cache size: due to limited cache capacity, once injected data are thrown out of the cache and injected again. In the system with large cache, there is no conflict, i.e., the elements of matrix B are injected only once during the execution. In addition, the efficiency of the cache injection decreases for longer MRC. In the system *B-64*, when MRC=20pclk, the increase of the processor number does not contribute to the speedup, since the execution time is completely determined by the bus traffic - the bus utilization is almost 100%. By increasing MRC, the bus utilization decreases (58.6% for the system *B-64* when MRC=100pclk). The cache injection dramatically reduces the bus traffic in both cases, but the relative improvement is larger when the bus utilization is higher in the base system.

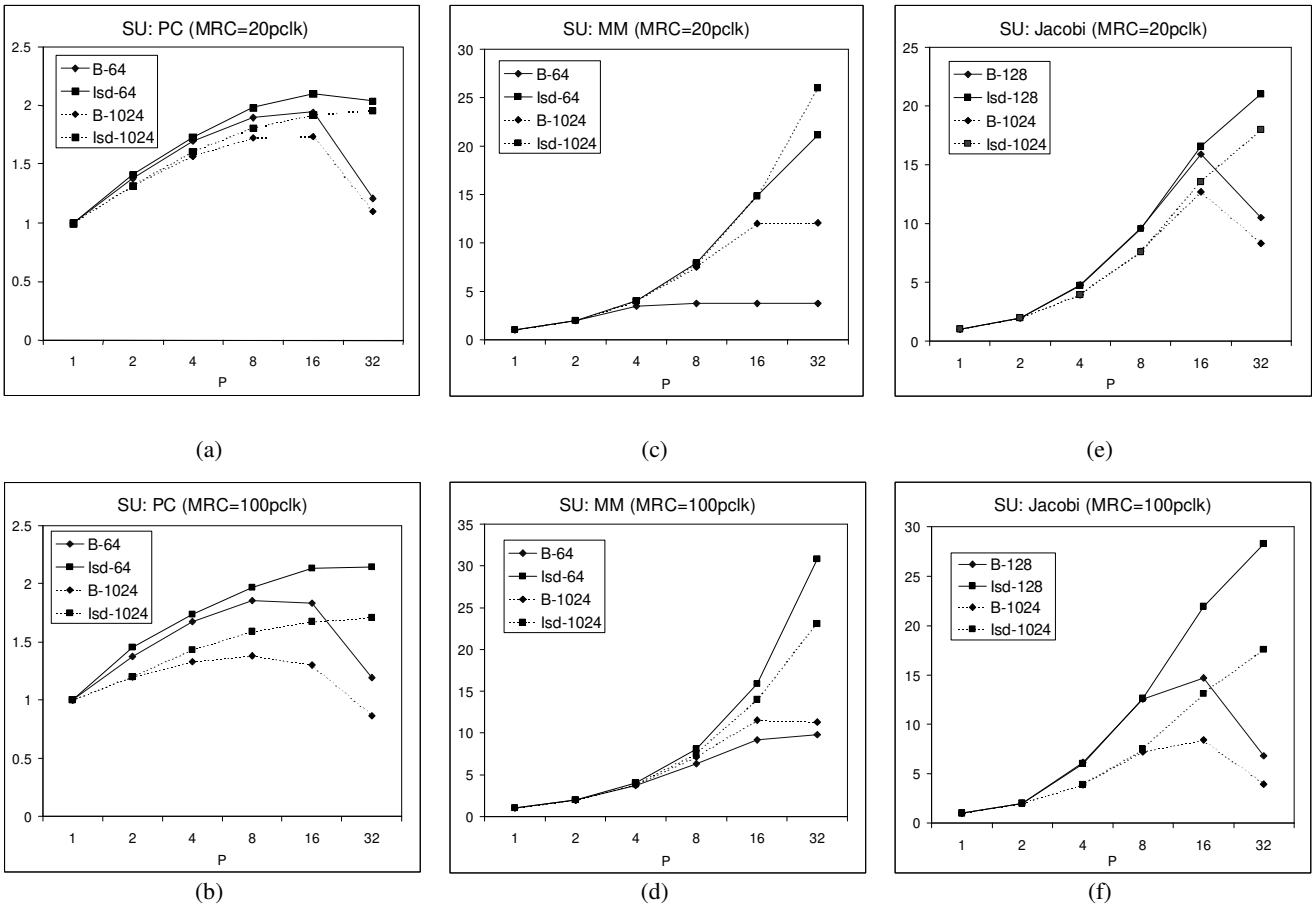

Figure 6. SU: PC, MM, and Jacobi.

Jacobi. Jacobi is a method for solving partial differential equations and iterates over a two-dimensional array. In each iteration, every matrix element is updated to the average of its four neighbors. A scratch array is used to store new values, in order to avoid overwriting an element's old value before it is used by its neighbor. All processors are assigned roughly equal chunks of rows. Neighboring processors share the rows on a chunk's boundary, so there is 1-Producer-1-Consumer sharing pattern. The matrix size is 256×256. Since this application has 1-Producer-1-Consumer sharing pattern, we have to apply the injection on write-back. A producer executes `Update` instructions, sending the data from boundary rows to neighbor processors. On the other hand, each consumer opens the address window to inject data from its neighbors.

When the number of processors in the system is small, with small cache memory size, the cache injection can degrade the performance negligibly (less than 2%). This is a consequence of the small percentage of shared data, the instruction overhead due to `Update` instructions, and conflicts between the injected data and the current working set. In all other cases, the injection improves performance, up to 77.7% (Figure 6e, f). The cache injection reduces the bus traffic during the synchronization and the miss rates for shared data (read bus cycles in *B* are replaced by software initiated write-back

bus cycles in *Isd*). The drop of the speedup in the base system with 32 processors is due to the high percentage of shared data and consequently, high bus traffic.

Radix. Radix sorts integer keys using the radix-sorting method. The algorithm is iterative, performing one three-phase iteration for each *r* digit of keys. First, each processor passes over its assigned keys and generates a local histogram. Next, local histograms are accumulated in a global histogram. Finally, each processor uses the global histogram to permute its keys into a new array for the next iteration. We use an 8-bit digit (r=256) to sort 128K keys. The injection of the global histogram *rank* is applied in the first phase of an iteration. Each processor initializes the injection table to accept the elements of *rank* array currently being updated by the next processor, which should insert an `Update` instruction after the last write in the cache block. In the second phase, each processor computes its *rank_ff*, using the global histogram *rank* and local histograms *rank_me* of all processors with lower ID. As there are multiple consumers, we use the injection on first read. Each processor initializes the injection table according to the data to be used at the beginning of the phase, and invalidates the entries at the end. In the last phase, there is an irregular communication all-to-all. If the size of cache block is one word, the cache injection significantly improves performance. Otherwise, the false sharing eliminates the benefit of injection, so we did not use the injection in this phase. Solutions *Is* and *Isd* always improve performance, and *Isd* outperforms *Is* (Figure 7a,b). Solution *Is* improves performance for up to 30.9%, while *Isd* improves for up to 41.1%.

LU. LU factors a dense matrix into the product of lower triangular and upper triangular matrices. The matrix is divided into blocks; a block ownership is assigned using a 2D-scatter decomposition, with blocks being updated by the processor that owns them. Outer loop iterates over the diagonal blocks. In the iteration *k,* first step is the factorization of the diagonal block A_{KK}. Next, processors modify the perimeter blocks in column *k* and row *k* using the diagonal block. Finally, processors modify the interior blocks using corresponding perimeter blocks. The matrix size is 512×512, and the block size is 8×8. In the second phase of the iteration *k*, the processors that own the perimeter blocks update those blocks, using the diagonal block A_{KK}, modified in the previous phase. As there are more consumers, each processor that owns a perimeter block inserts instructions to support the injection of the diagonal block. In the third phase, the processors modify the interior blocks, using the corresponding perimeter blocks. In this phase, there are also more consumers, so at the beginning of the phase each processor inserts the instructions to support the injection of the corresponding perimeter blocks. It should be noted that here we can use the injection on write-back. In this case, the process of injection would be translated into the first phase for the diagonal block A_{KK}, and to the second phase for the perimeter blocks.

Solution *Is* always improves performance, while solution *Isd* does not influence the execution time in the systems with 2 and 4 processors (Figure 7c, d). Surprisingly, *Is* always outperforms *Isd*, except when P=32, MRC=20pclk, and cache size 128KB, although Isd reduces cache misses and bus traffic more than *Is*. The main reason why this potential did not turn into the performance gain is due to the execution of the inserted instructions supporting the injection of matrix blocks, and the contention on internal resources of cache controller caused by these instructions and injection mechanism. The large instruction overhead is due to the complex initialization of the injection table for a matrix block, since it does not occupy continual address space. This problem could be easily solved by the data reallocation.

FFT. FFT executes the 1-D version of the six-step FFT algorithm. The data set consists of the *n* complex data points to be transformed, and *n* complex data points referred as the *roots of unity*. Both sets of data are organized as $\sqrt{n} \times \sqrt{n}$ matrices, which are partitioned among processors in contiguous chunks of rows. The only communication is during three transpose phases. The matrices size is 256×256. In the algorithm steps 2, 3, and 5, each processor modifies only its assigned chunk of rows. In the steps 1, 4, and 6, the matrix is transposed: the processor communication is all-to-all, and the data sharing pattern is 1-Producer-1-Consumer. A producer inserts `Update` instructions before the transposing step, while a consumer initializes the injection table to inject the corresponding data.

Solution *Is* always improves performance, while *Isd* degrades performance in the systems with cache size 128KB and MRC=20pclk, between 3% when P=2 and 0.4% when P=16 (Figure 7e, f). In the systems with small cache, small memory read cycle time and moderate number of processors, solution *Is* outperforms *Isd*, due to the conflicts in caches between the injected data and the current working set. The problem of complexity of the initialization is the same as in LU. Solution *Is* improves performance up to 13.3%, and *Isd* up to 15.8%.

Ocean. Ocean simulates large-scale ocean movements. At each horizontal cross-section through the ocean basin, several different variables are modeled. Each variable is discretized and represented by a regular, uniform two-dimensional grid with *n*×*n* non-border points. After the initialization, this application proceeds over a large number of time-steps. Data are partitioned among processors in square-like subgrids. Most of time the application solves partial differential equations using the red-black Gauss-Seidell equation solver. The grid size is 128×128. The injection of true shared data is

implemented only in the phase of the solving of partial differential equations. Generally, a processor communicates with four neighbor processors (Top, Bottom, Left, Right); the data sharing pattern is 1-Producer-1-Consumer. A producer initiates update of the consumer cache with data to be used in the next iteration. A consumer initializes the injection table to accept the last row of subgrid assigned to the processor Top, first row of the Bottom, left column of the Right and right column of the Left.

Solution *Is* always improves performance, while *Isd* degrades performance in the systems with P=2 and P=4, cache size 128KB and MRC=20pclk (Figure 7g, h). Solution *Isd* does not contribute to the performance improvement, compared to *Is*, which is simpler to implement. Although *Isd* solution reduces cache misses and bus traffic, complexity of the inserted code and the contention on the internal resources limit the benefit of these solutions. The problem of the complexity of the initialization is the same as in LU. Solution *Is* improves performance up to 90.9%, and *Isd* up to 89.6%. For this problem size of the application, the base solution shows poor speedup, due to the high communication-to-computation ratio and high bus utilization.

5. Related Work

Flexibility and efficiency of software-controlled techniques for reducing memory latency, and the ever increasing speed gap between memory subsystem and high performance processors in cache coherent shared memory systems result in a large number of research studies. This section gives a short overview of recent research efforts in this area [9].

T. Mowry proposed a compiler algorithm for selective prefetching in uniprocessor and multiprocessor systems [10]. Simulation analysis based on DASH-like CC-NUMA multiprocessors showed considerable improvements, between 6% and 53% reduction in execution time. However, the proposed algorithm is limited to numeric-based applications. T. Mowry and C. Luk proposed three different prefetching schemes for applications with pointer-based data structures [11]. Performance analysis shows performance improvement of 14% for the Barnes application from the SPLASH-2 benchmark suite. P. Ranganathan et al. investigated the interaction of software prefetching with ILP processors in shared memory systems [14]. D. Tullsen and S. Eggers analyzed the potential of "ideal" compiler-directed prefetching in bus-based shared memory multiprocessors [18]. Results have shown that despite high memory latency, bus based architectures are not very well suited for prefetching, predominantly due to the invalidation misses and the increase of bus traffic. Same authors proposed several techniques and heuristics that increased efficiency of prefetching for these architectures [19].

Data forwarding technique was implemented for the first time in the DASH multiprocessor, developed at Stanford University [6]. Special `deliver` instruction initiates the sending of a cache block to processors specified by a bit vector. Koufaty et al. proposed a framework for a compiler algorithm to insert data forwarding instructions in the code that exploits loop-level parallelism with do-all constructs [5]. Simulation analysis based on CC-UMA architecture has shown performance improvement of 50% for a system with large caches and 30% for a system with small caches. In CC-NUMA architectures with write-invalidate protocols, it may be useful to update the memory copy of the shared data, if another processor is going to use that data. J. Skeppstedt and P. Stenstrom proposed a compiler algorithm for inserting the instructions to update a memory block after its last modification, using a classic data-flow analysis [16].

Large number of research papers investigates the combination of prefetching and forwarding techniques [1, 13, 15, 17]. H. Shafi et al. evaluated fine-grain producer-initiated communication, alone and combined with data prefetching [15]. P. Trancoso and J. Torrellas used data prefetching and data forwarding to speed up critical sections [17]. U. Ramachandran et al. explored the set of software-controlled primitives, allowing selective update of cache memories, and generalized data prefetching [13]. G. Byrd and M. Flynn classified communication mechanisms in shared memory multiprocessors [1].

Dahlgren analyzed the deficiencies of hybrid snooping cache protocols and the effectiveness of read snarfing and write cache in boosting the performance of these protocols [3]. In read snarfing (or read broadcast) data block that is transferred on the bus as a read response, updates the node that requested it, but also updates all other caches having the block invalidated. This technique and cache injection use similar mechanism, but cache injection is more flexible since it is software-controlled.

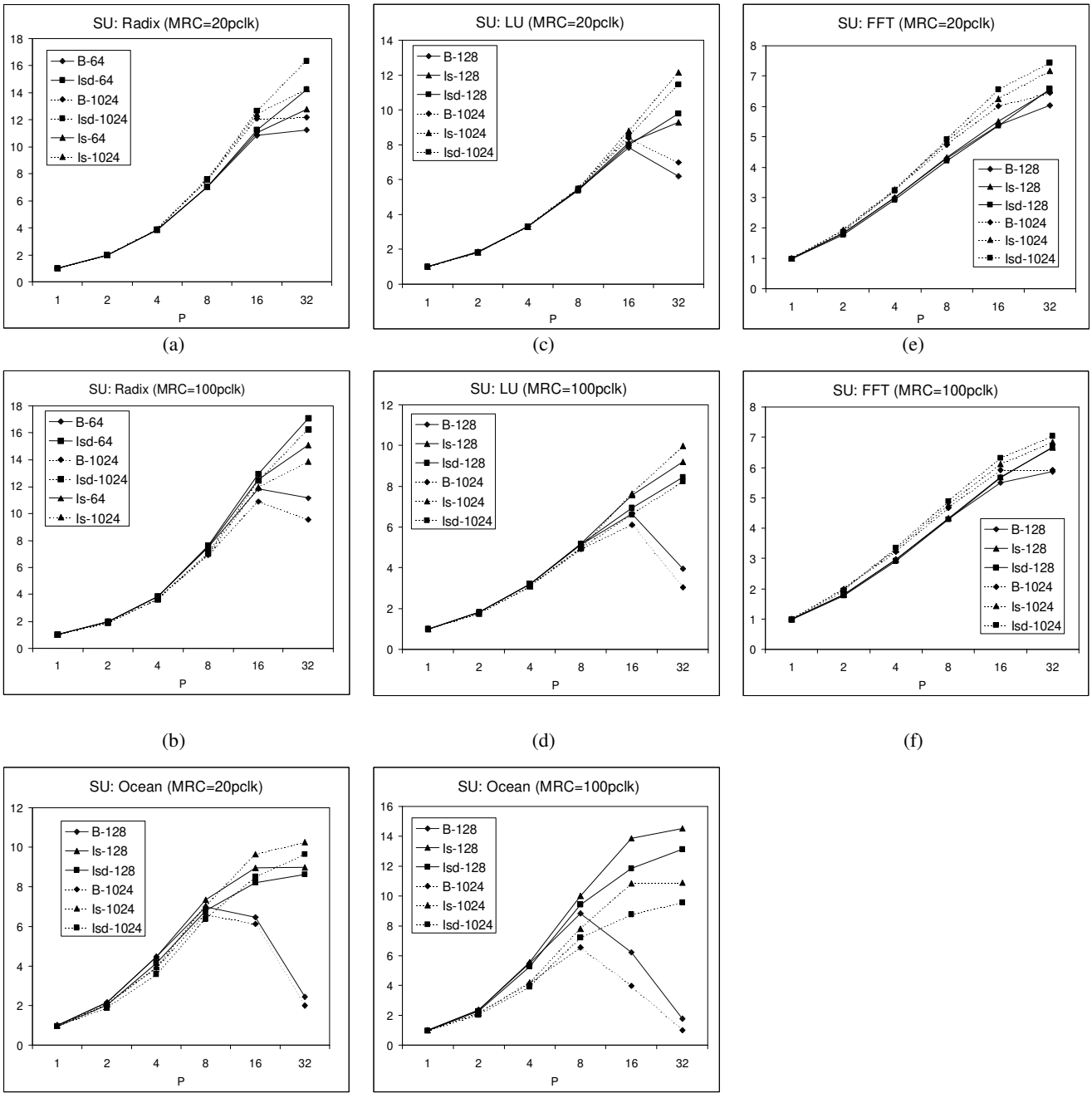

Figure 7. SU: Radix, LU, FFT and Ocean.

6. Conclusions

This paper presents a novel software controlled technique for tolerating memory latency in bus-based shared memory multiprocessors called cache injection. Cache injection is developed to overcome some of the shortcomings of the existing software-controlled techniques, cache prefetching and data forwarding, combining advantages of these two techniques and inherent characteristics of bus-based architectures. Experimental analysis, based on execution driven simulation, showed gains between 12% and 96% over the based system for synchronization kernels, and between 2% and 90% for applications. Performance improvements are due to the elimination of cache misses for true shared data and reduction of bus traffic.

Efficiency of cache injection increases with the increase of the number of processors in the system, cache size, and memory latency.

Possible future research includes developing and implementation of a compiler algorithm for inserting instructions to support injection of shared data. It is also interesting to analyze the effectiveness of cache injection compared to data prefetching, read-snarfing, software-controlled update, and the combinations of these techniques. Another direction is to implement some kind of cache injection in scalable cache coherent shared memory multiprocessors [8].

References

[1] Byrd G. T., Flynn M. J., "Producer-Consumer Communication in Distributed Shared Memory Multiprocessors," *Proceedings of the IEEE*, vol. 87, no. 3, March 1999, pp. 456-466.

[2] Culler D., Singh J. P., Gupta A., *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann Publishers, San Francisco, CA, August 1998.

[3] Dahlgren, F., "Boosting the Performance of Hybrid Snooping Cache Protocols,", *Proceedings of the 22nd ISCA*, June 1995, pp. 60-69.

[4] Dahlgren, F., Skeppstedt, J., Stenstrom, P., "Effectiveness of Hardware-Based and Compiler-Controlled Snooping Cache Protocol Extensions," *Proceedings of the HiPC*, December 1995, pp. 87-92.

[5] Koufaty D. A., Chen X., Poulsen D. K., Torrellas J., "Data Forwarding in Scaleable Shared Memory Multiprocessors," *IEEE Transactions on Parallel and Distributed Technology*, Vol. 7, No. 12, 1996, pp. 1250-1264.

[6] Lenoski D., Laudon J., Gharachorloo K., Weber W., Gupta A., Hennessy J., Horowitz M., Lam M. S., *The Stanford DASH Multiprocessor*, IEEE Computer, March 1992, pp. 63-79.

[7] Magdic, D., "Limes: A Multiprocessor Simulation Environment," *TCCA Newsletter*, March 1997, pp. 68-71.

[8] Milenkovic A., Milutinovic V., "Lazy Prefetching," *Proceedings of the 31st HICSS*, IEEE Computer Society Press, Vol. 7, January 1998, pp. 780-782.

[9] Milenkovic A., "Achieving High Performance in Bus-Based Shared Memory Multiprocessors, " *IEEE Concurrency*, Vol. 8, No. 3, July-September 2000, pp. 36-44.

[10] Mowry T., *Tolerating Latency Through Software-Controlled Data Prefetching*, Phd Thesis, Stanford University, 1994.

[11] Mowry T, Luk C., "Predicting data Cache Misses in Non-Numeric Applications Through Correlation Profiling," *Proceedings of the 30th Micro*, December 1997, pp. 314-320.

[12] Protic, J., Tomasevic, M. and Milutinovic, V., *Distributed Shared Memory: Concepts and Systems*, IEEE Computer Society Press, Los Alamitos, California, 1998.

[13] Ramachandran U., Shah G., Sivasubramaniam A., Singla A., Yanasak I., "Architectural Mechanisms for Explicit Communication in Shared Memory Multiprocessors," *Proceedings of the Supercomputing'95*, vol. 2, December 1995, pp. 1737-1775.

[14] Ranganathan P., Pai V., Abdel-Shafi H., Adve S., "The Interaction of Software Prefetching with ILP Processors in Shared-Memory Multiprocessors," *Proceedings of the 24th ISCA*, June 1997, pp. 144-156.

[15] Shafi H. A., Hall J., Adve S., Adve V., "An Evaluation of Fine-Grain Producer Initiated Communication in Cache-Coherent Multiprocessors," *Proceedings of the 3rd HPCA*, February 1997, pp. 204-215.

[16] Skeppstedt J., Stenstrom P., "A Compiler Algorithm that Reduces Read Latency in Ownership-Based Cache Coherence Protocols," *Proceedings of the PACT'95*, IEEE Computer Society Press, June 1995, pp. 69-78.

[17] Trancoso P., Torrellas J., "The Impact of Speeding up Critical Sections with Data Prefetching and Forwarding," *Proceeding of the 25th ICPP*, IEEE Computer Society Press, Vol. 3, August 1996, pp. 79-86.

[18] Tullsen D., Eggers S., "Limitations on Cache Prefetching on a Bus-Based Multiprocessor," *Proceedings of the 20th ISCA*, June 1995, pp. 392-403.

[19] Tullsen D., Eggers S., "Effective cache prefetching on bus-based multiprocessors," *ACM Transactions on Computer Systems*, Vol. 13, No. 1, 1995, pp. 57-88.

[20] Woo S. C., Ohara M., Torrie E., Singh J. P., Gupta A., "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proceedings of the 22nd ISCA*, June 1995, pp. 24-36.