Cache Injection: A Novel Technique for Tolerating Memory Latency in Bus-Based SMPs

Aleksandar Milenkovic, Veljko Milutinovic

School of Electrical Engineering, University of Belgrade,
P. Box 35-54, 11120 Belgrade, Yugoslavia
{emilenka, vm}@etf.bg.ac.yu

Abstract. Cache misses and bus traffic are key obstacles to achieving high performance of bus-based shared memory multiprocessors using invalidation-based snooping caches. To overcome these problems, software-controlled techniques for tolerating memory latency can be used, such as cache prefetching and data forwarding. However, some previous studies have shown that cache prefetching is not so effective in bus-based shared memory multiprocessors, while data forwarding is not easy to implement in this environment. In this paper, we propose a novel technique called cache injection, which combines consumer and producer initiated approaches, as well as the broadcasting nature of bus. Performance evaluation based on program-driven simulation and a set of eight parallel benchmark programs shows that cache injection is highly effective in reducing coherence misses and bus traffic.

1 Introduction

Private caches are essential to reduce the bus traffic and the memory latency in bus-based shared memory multiprocessors (SMPs). In such systems, snooping write-invalidate cache coherence protocols are commonly accepted as an effective approach to keep the data coherent [1]. However, the problem of high memory latency is still the most critical performance issue in these systems. One way to cope with this problem is to tolerate high memory latency by overlapping memory accesses with computation. The importance of techniques for tolerating high memory latency in multiprocessor systems increases, due to the widening speed gap between CPU and memory, high contention on the bus, bus traffic caused by data sharing between processors, and the increasing physical distances between processors and memory.

Software-controlled cache prefetching is a widely accepted consumer-initiated technique for tolerating memory latency in multiprocessors, as well as in uniprocessors. In software-controlled cache prefetching, a CPU executes a special `prefetch` instruction that moves a data block (expected to be used by that CPU) into its cache, before it is actually needed [2]. In the best case, the data block arrives at the cache before it is needed, and the CPU load instruction results in a hit. However, 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. With data forwarding, when a CPU produces the data, 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 [3]. Therefore, when consumer processors access the data block, they find it in their caches.

Most of the studies [2-8] examined the effectiveness of cache prefetching and data forwarding in CC-(N)UMA architectures, except [9], which examined the potential of cache prefetching in bus-based SMPs. This study reported poor effectiveness of cache prefetching, despite the 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, it can result in performance degradation. Second, too early initiated prefetching can negatively affect data sharing. Last, current prefetching algorithms are not so effective in predicting coherence misses. Actually, coherence misses represent the biggest challenge for designers, especially as caches become larger and they dominate the performance of parallel programs.

On the other side, 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 SMPs, where a special instruction initiates an update of all invalid copies of the specified cache block in the system [10]. 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, it is less flexible than classic data forwarding as defined in [2], because it does not allow forwarding to the processors not having the invalid copies of the data block. In paper [11], Anderson and Baer showed that the technique called read snarfing could be very effective in reducing the number of coherence misses and the bus traffic in bus-based SMPs. With read snarfing, a data block that is transferred on the bus as a read response not only updates the node that requested it, but also updates all other caches having the block invalidated. Read snarfing is a hardware-based technique, easy to implement. However, it is based on the heuristic that all blocks that are invalid will be needed in the future, and its effectiveness highly depends on cache size. In the system with relatively small cache size, the invalid cache blocks will be probably displaced from the cache, so read snarfing is not applicable.

In this paper, we propose a novel software-controlled technique called cache injection, aimed to reduce coherence misses and bus traffic. Using advantages of the existing techniques and the characteristics of bus-based architectures, cache injection overcomes some of the shortcomings of the existing techniques, such as: (a) bus and memory contention, (b) negative impact on data sharing and instruction overhead in the case of cache prefetching, and (c) compiler and implementation complexity in case of data forwarding. The proposed technique can be combined with the existing ones in order to raise performance in bus-based SMPs.

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

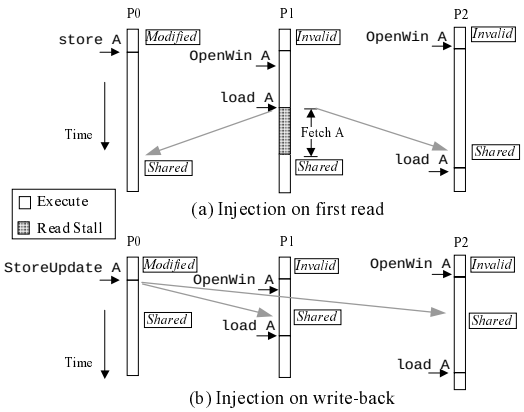
2 Cache Injection

In cache injection, a consumer predicts its future needs for shared data by executing an `OpenWin` instruction. This instruction only stores the first and the last address of successive cache blocks, in a special local injection table. This address scope is called address window. There are two main scenarios when cache injection could happen: during the read bus transaction (injection on first read) or during the 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, it sees 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 processor stores the block into its cache (Fig. 1a). Hence, in case of multiple consumers, only one read bus transaction is needed to update all consumers, if they all have initialized their injection tables.

Injection on write-back bus transaction is applicable when shared data exhibit both 1-Producer-1-Consumer (1P-1C) and 1-Producer-Multiple-Consumers (1P-MC) patterns. In these scenarios, each consumer also initializes its injection table. At the producer side, after the data producing is finished, the producer initiates write-back bus transactions in order to update the memory, by executing an `Update` instruction. During this transaction, all consumers snoop the bus, and if they find injection hit, they catch the data block from the data bus and store it into their caches (Fig. 1b).

The above definition of cache injection assumes a bus-based SMP where each processor has one or more levels of cache memory and a write-back invalidate cache-coherence protocol based on snooping. Hardware support for cache injection includes injection table, proposed instructions (Fig 1c), and a negligible modification of the bus control unit. The 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.



`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 [7].

`StoreUpdate <Addr><Value>` - performs an ordinary store instruction; in addition, it initiates a write-back bus transaction [7].

(c) Proposed instructions

Fig. 1. Cache injection mechanism.

3 Experimental Methodology

We evaluate the performance impact of cache injection using Limes [12] – a tool for program-driven simulation of SMPs. A synchronization kernel (LTEST), three parallel test applications well suited to demonstrate various data sharing patterns (PC, MM, Jacobi), and four applications from SPLASH-2 suite (Radix, FFT, LU, Ocean) [13] are used in the evaluation. They are all written in C using the ANL macros to express parallelism and compiled by gcc with the optimization flag –O2. Proposed instructions for cache injection support are hand-inserted into the applications. For each application we compare the number of read misses and the bus traffic for the base system (B), the system with read snarfing (S), the system with software-controlled update and read snarfing (U), and the system with cache injection (I).

The modeled architecture is bus-based SMP containing 16 processors 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 first level cache memory. We assume that instructions always hit into the cache. Cache hit is solved without penalty. The relevant system parameters are the following: cache line size is 32B, data bus width is 8B, snoop cycle is 2pclk (pclk - processor cycle), and write-back buffer size is 32B. The read and the read-exclusive bus transactions include the request and the response phases. The memory read cycle defines time needed to retrieve a requested block from memory; assumed value is 20pclk. A two-word transfer via the data bus takes 2pclk; 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. A 128-entry injection table was used in the evaluation.

We have used the following data sets: 1000 acquire requests per processor for LTEST, 128×128 shared matrix and 20 iterations for PC, 128×128 matrix for MM, 256×256 matrix and 20 iterations for Jacobi, 128K keys with 8-bit digit for Radix, 256×256 matrix with 8×8 blocks, 256×256 for FFT, and 130×130 for Ocean.

The aim of our evaluation is to first 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. 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. 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 a 1P-MC 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 1P-1C, we have to use injection on write-back.

4 Results

For synchronization kernel LTEST both read snarfing and cache injection are highly effective: read snarfing reduces the number of read misses and the bus traffic for 90% and 88%, respectively, while cache injection for 92% and 90%. Since the effectiveness of these two techniques is approximately the same for synchronization operation, we do not model synchronization requests on the bus in the experiments with the parallel applications. In this way, we avoid the over-estimation of the synchronization overhead due to relatively small data sets.

Fig. 2 shows the number of read misses and the bus traffic for parallel applications, normalized to the base system, when the caches are relatively small (left) and relatively large (right). For all applications cache injection (I) outperforms read snarfing (S) and software-controlled update with read snarfing (U). The effectiveness of solution I relative to solutions S and U is higher in the system with small caches: invalid blocks are frequently displaced from the cache and in that case snarfing is not applicable. Next, cache injection can be effective in reducing cold misses, when there are multiple consumers of shared data, while snarfing can eliminate only coherence misses. Last, cache injection increases the possibility of successful injection, since the time window during which a block can be injected is software-controlled. The rest of this section explains the data sharing patterns and injection support, and discusses results for each application.

PC. In PC, the coherence misses dominate since each processor modifies its assigned sub matrix, which is read by all other processors in the next iteration (1P-MC sharing pattern). Solutions S and U are almost as effective as cache injection, in the system with large caches. Slight advantage of cache injection is due to elimination some of cold misses. However, in the system with small caches, solutions S and U are not effective at all. The main reason for this is that invalidated data, which should be updated during the next bus read or write-back transaction, are displaced from the cache due to cache conflicts.

MM. MM is a parallel version of matrix multiplication A=AxB, where each processor computes elements of the assigned sub matrix of matrix A. As all processors only read elements of the shared matrix B, to support cache injection each processor defines an address window encompassing the whole matrix B. Cache injection reduces the number of read misses and bus traffic for 92% and 88%, respectively, in the system with small caches, and for 91% and 77% in the system with large caches. Here solutions S and U are not effective at all since the shared data are read only predominantly. The efficiency of cache injection does not increase as the cache size increases. The system with small caches exploits the benefit of multiple injections of data which are thrown out of the cache due to cache conflicts, while in the system with large caches the elements of matrix B are injected only once during the execution.

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. All processors are assigned roughly equal chunks of rows. Neighboring processors share the rows on a chunk's boundary, so there is a predominantly 1P-1C sharing pattern. Consequently, we have to apply the injection on write-back. Solution S is not effective at all, while solutions U and I are equally effec-

tive and reduce the number of read misses for 47% in the system with large caches; in the system with small caches solution I is slightly more effective.

Radix. Radix sorts integer keys using the three-phase iterative radix-sorting method. The injection of the global histogram *rank* is applied in the first phase of 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. In the last phase, there is an irregular all-to-all communication, so we did not use the injection in this phase. In the system with small caches, solutions S, U, and I reduce the number of read misses for 7%, 8%, and 21%, and the bus traffic for 4%, 3%, and 9%, respectively. In the system with large caches, they reduce the number of read misses for 18%, 21%, and 26%, and the bus traffic for 10%, 10%, and 12%, respectively.

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 organized as $\sqrt{n} \times \sqrt{n}$ matrices, which are partitioned among processors in contiguous chunks of rows. 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 1P-1C. A producer inserts `Update` instructions before the transposing step, while a consumer initializes the injection table to inject the corresponding data. Solution S is not effective since there is predominantly 1P-1C sharing pattern. In the system with small caches, the effectiveness of solutions U and I is limited by conflicts in caches; the number of read misses is reduced for 3% and 8%, respectively, while the bus traffic is increased for 11%, and 7%, respectively. In the system with large caches, solution I is highly effective and reduces 46% of read misses, and 12% of bus traffic, while solution U reduces 30% of read misses and 1% of bus traffic.

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 2D-scatter decomposition, with blocks being updated by the processor that owns them. Outer loop iterates over the diagonal blocks. 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 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. Solution I outperforms solutions S and U; it reduces the number of read misses and bus traffic for 30% and 22%, respectively, in the system with small caches, and for 38% and 31% in the system with large caches.

Ocean. Ocean simulates large-scale ocean movements. Data set consists of the uniform two-dimensional grids with $n \times n$ non-border points, partitioned among processors in square-like sub grids. Most of time, the application solves partial differential equations using the red-black Gauss-Seidell equation solver. The injection of true-shared

data is implemented predominantly 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 1P-1C. 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 the sub grid assigned to the processor Top, first row of the Bottom, left column of the Right and right column of the Left. In the system with small caches, solutions S, U, and I reduce the number of read misses for 14%, 17%, and 25%, and the bus traffic for 19%, 16%, and 28%, respectively. In the system with large caches, solutions S, U, and I reduce the number of read misses for 28%, 35%, and 48%, and the bus traffic for 34%, 30%, and 44%, respectively.

Additional experiments not presented in this paper, which varied architectural parameters, show that the efficiency of cache injection increases with 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 the overall miss rate and reducing the bus traffic. Larger caches reduce 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.

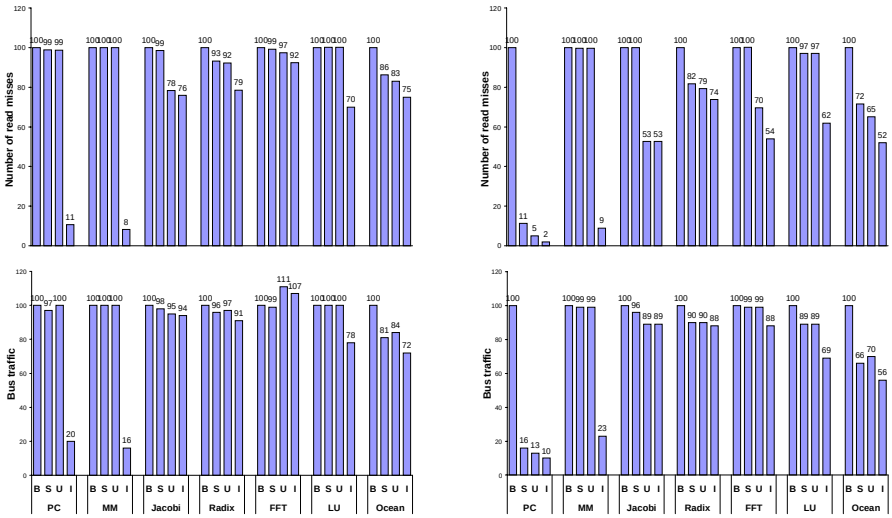


Fig. 2. Number of read misses (upper) and bus traffic (lower) relative to the base system. Cache_size=64/128KB (128KB for FFT, LU, Ocean) (left), and Cache_size=1024KB (right).

5 Conclusion

This paper presents a novel software-controlled technique for tolerating memory latency in bus-based SMPs. This technique, called cache injection, has been developed in order to overcome some of the shortcomings of the existing techniques, cache pre-

fetching, software-controlled update, and read snarfing, combining advantages of these techniques and inherent characteristics of bus-based architectures.

Experimental analysis, based on execution driven simulation, showed highly effectiveness of cache injection in reduction of the number of read misses and the bus traffic, compared to the base system. In addition, it provides further improvements compared to the systems with read snarfing and software-controlled update.

Possible future research includes developing and implementation of a compiler algorithm for inserting instructions to support injection of shared data. Another direction is to implement some kind of cache injection in scalable cache coherent shared memory multiprocessors.

References

1. Culler D., Singh J. P., Gupta A.: Parallel Computer Architecture: A Hardware/Software Approach, Morgan Kaufmann Publishers, San Francisco, CA (1998)
2. Mowry T.: Tolerating Latency Through Software-Controlled Data Prefetching. Ph. D. Thesis, Stanford University, (1994)
3. 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) 1250-1264
4. Byrd, G. T., Flynn M. J.: Producer-Consumer Communication in Distributed Shared Memory Multiprocessors. Proceedings of the IEEE, vol. 87, no. 3. (1999) 456-466
5. 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. (1995), 1737-1775
6. 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. (1997) 204-215
7. 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. (1995) 69-78
8. 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. (1996) 79-86
9. Tullsen D., Eggers S.: Effective cache prefetching on bus-based multiprocessors. ACM Transactions on Computer Systems, Vol. 13, No. 1. (1995) 57-88
10. Dahlgren, F., Skeppstedt, J., Stenstrom, P.: Effectiveness of Hardware-Based and Compiler-Controlled Snooping Cache Protocol Extensions. Proceedings of the HiPC. (1995) 87-92
11. Anderson, C., Baer, J.-L.: Two Techniques for Improving Performance on Bus-Based Multiprocessors. Proceedings of the 1st HPCA. (1995) 256-275
12. Magdic, D.: Limes: A Multiprocessor Simulation Environment. TCCA Newsletter, March 1997. 68-71
13. 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, (1995) 24-36