Cache Injection on Bus Based Multiprocessors

Aleksandar Milenkovic, Veljko Milutinovic
School of Electrical Engineering, University of Belgrade
E-mail: {emilenka,vm}@etf.bg.ac.yu, Http: {galeb.etf.bg.ac.yu/~vm, ubbg.etf.bg.ac.yu/~emilenka}

Abstract

Software-controlled cache prefetching and data forwarding are widely used techniques for tolerating memory latency in shared memory multiprocessors. However, some previous studies show that cache prefetching is not so effective on bus-based multiprocessors, while the effectiveness of data forwarding has not been explored in this environment, yet. In this paper, a novel technique called cache injection is proposed. Cache injection, tuned to the properties of bus-based architectures, combines advantages of both cache prefetching and data forwarding. Some preliminary experiments show that the proposed solution can significantly help in reducing the overall miss ratio and bus traffic in applications where write-shared data prevails.

1. Introduction

The problem of high memory latency is the most critical performance issue in the cache-coherent shared memory multiprocessors. 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 several reasons: (a) the widening gap in speed between CPU and memory, (b) high contention on interconnection networks, (c) interconnect operations, caused by data sharing between processors, and (d) 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, the CPU executes a special prefetch instruction that moves data, expected to be used by that CPU, into its cache, before it is actually needed. In the best case, the data arrives at the cache before it is needed by the CPU, and the CPU sees its load as a hit. However, prefetching is inapplicable or insufficient for some communication patterns such as irregular communication, synchronization, producer-consumer sharing patterns, etc. In addition, prefetching is not without costs and it can negatively affect data sharing, overall cache miss rates, and contention on interconnection network.

For many programs and sharing patterns (e.g., producer-consumer), producer-initiated data transfers are a natural style of communication. In literature, producer initiated primitives are known as data forwarding, delivery, remote writes, and software-controlled updates. With data forwarding, when the 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. Therefore, when the consumer processors access the data, they find it in their caches.

Although numerous evaluation studies showed that prefetching and forwarding are highly effective in CC-NUMA multiprocessors, their effectiveness on bus-based multiprocessors is explored much less or not enough. Some previous studies show that bus-based multiprocessors are not very well suited for prefetching, despite high memory latency, while the effectiveness of data forwarding in bus-based multiprocessors has not been explored enough, yet.

In this paper, a novel solution called cache injection is proposed, aimed to rise the effectiveness of techniques for tolerating memory latency on bus-based multiprocessors, is proposed. This technique combines the good properties of both, cache prefetching and data forwarding, and the appropriateness of bus-based architectures. In cache injection, a consumer predicts its future needs using a **`prefetch`**-like instruction. However, instead to initiate the read bus transaction, this instruction only puts the address of the data expected to be used in the special *injection table*, which is a part of the cache controller. The producer, after the data is produced, executes a **`write_back`** instruction, which initiates the write-back bus transaction; all consumers snoop the bus, and if they find the *injection table* hit, they catch the data from the bus and store it into their caches.

The rest of this paper is organized as follows. Section 2 discusses related work. Motivation for this research is given in Section 3. Section 4 describes the proposed

solution, discusses its implementation, and demonstrates its benefit using a simple example code. Section 5 describes the experimental methodology used for performance evaluation and gives some preliminary results. Section 6 gives some concluding remarks.

2. Related Work

Software-controlled cache prefetching and data forwarding, as the most promising techniques for tolerating high memory latency in shared memory multiprocessors, have been studied by many researchers. A short description of the relevant studies, given in this section, is aimed to give the background and to better explain the motivation for our research.

In software-controlled prefetching, the analysis of what to prefetch and the scheduling of when to initiate prefetches are two key steps done statically by software. The responsibility for inserting prefetch instructions is on compiler or programmer. Mowry developed the most sophisticated compiler algorithm for prefetching [1]. Performance evaluation based on numerous architectural simulations of CC-NUMA multiprocessors showed that this loop-based algorithm is quite successful at hiding memory latency, improving performance of some scientific applications by as much as twofold.

Tullsen and Eggers evaluated software-controlled prefetching on bus-based multiprocessors [2]. The results showed that bus-based architectures are not well suited for prefetching: for several variations of the architecture, speedups for chosen benchmarks were no greater than 39%, and the degradations were as high as 7%, when prefetching was added to the workload. They identified memory and bus contention, and data sharing as major factors responsible for such behavior.

Ranganathan et al. evaluated the effectiveness of software-controlled prefetching in shared memory multiprocessors built of the state-of-the-art processors which exploit instruction level parallelism (ILP) [3]. They found that software prefetching results in significant reductions in execution times. However, compared to previous-generation systems, software prefetching is significantly less effective in reducing memory stall component of execution time on an ILP-based system.

Koufaty et al. developed a framework for a compiler algorithm for forwarding [4]. They focused on the code that exploits loop-level parallelism with doall constructs and array accesses that are indexed by affine functions of the loop indices and constants. Performance evaluation based on trace-driven simulation of a CC-UMA multiprocessor showed that data forwarding provides considerable speed up for PerfectClub parallel applications: 30-50% depending on cache size.

Abdel-Shafi et al. evaluated the performance impact of fine-grained producer-initiated communication, both with and without software prefetching in shared memory multiprocessors [5]. They used simple heuristics based on application behavior to insert prefetches and primitives that provide producer-initiated communication (**`WriteSend`** and **`WriteThrough`**). Simulation analysis of a CC-NUMA multiprocessor showed that these primitives provide additional benefits over prefetching for irregular communications and producer-consumer sharing patterns that are not suitable for software prefetching.

Trancoso and Torellas reduced time taken by critical sections by using software-controlled prefetching and forwarding to minimize the number of misses inside critical sections [6]. Skeppstedt and Stenstrom proposed a method that shortens read-miss latency in CC-NUMA multiprocessors with write-invalidate protocols [7]. They developed a compiler algorithm that replaces last store instruction to a memory block by an update instruction, using classical dataflow analysis techniques.

3. Motivation

Comparing prefetching and forwarding, it should be noted that prefetching can eliminate any kind of read misses (cold, conflict, and coherence), while forwarding can eliminate only coherence misses and the related case of cold misses. Prefetching is not effective in the cases when the value to be read is not produced sufficiently early, or when the location to be accessed is not known sufficiently early. Too early issued prefetch can degrade performance. For the coherence misses forwarding can be more effective than prefetching due to the following reasons: (a) forwarding delivers data to consumers as soon as it is produced, (b) smaller latency, and (c) possibility to forward the same data to several consumers with a single instruction. However, compiler support for data forwarding must be more sophisticated than 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.

Majority of previously mentioned studies examined CC-NUMA or CC-UMA multiprocessor architectures, except [2], which examined software-controlled cache prefetching on bus-based shared memory multiprocessors. In addition, we are not aware of any research evaluating producer-initiated communication primitives on bus-based multiprocessors. The study [2] reported poor effectiveness of software prefetching on

bus-based multiprocessors. There are three main reasons for this behavior. First, prefetching attempts to increase processor utilization by lowering the CPU miss rate. A smaller CPU miss rate is usually achieved at the expense of total miss rate, which increases memory traffic. However, bus-based architectures are more sensitive to changes in memory traffic, so that higher memory traffic can result in performance degradation. Next, prefetching can negatively affect data sharing when the future working set for one processor is in conflict with the current working sets of other processors. Last, current prefetching algorithms are not so effective in predicting invalidation misses. Sharing misses represents the biggest challenge for designers, particularly as caches become larger and invalidation misses dominate the performance of parallel programs.

In this paper we propose a novel technique called cache injection, aimed to rise the effectiveness of the existing software-controlled techniques cache prefetching and data forwarding on bus-based shared memory multiprocessors. Using advantages of both cache prefetching and data forwarding and the suitability of bus-based architectures, cache injection should overcome some of the shortcomings of the existing algorithms for cache prefetching and data forwarding, such as: bus and memory contention, negative impact on data sharing (too early or too late issued prefetch), compiler complexity, and instruction overhead. The proposed solution can be combined with the existing ones in order to rise overall effectiveness of techniques for tolerating memory latency in bus-based multiprocessors.

4. Cache Injection

4.1. Definition and Programming Model

In the cache injection [8, 9], a consumer predicts its future needs for write shared data using **`lprefetch`** instruction (lazy prefetch) instead of classical prefetch. In the case of cache miss, this instruction does not initiate bus transactions as in classical prefetching, but only puts the address of the requested block to the *injection table*. At the producer side, after the data producing is finished, the producer initiates bus transactions in order to update the memory, by executing a **`write_back`** instruction. During the write-back bus transaction, all consumers snoop the bus, and if they find that the current bus address belongs to some of the opened address window in the *injection table*, they catch the data from the data bus and store it into their caches. Hence, if the sharing pattern follows the *Single-Writer-Multiple-Reader* paradigm, only one bus transaction is needed to update all consumers, if the prediction was successful. Unlike classical prefetching, this approach can not cause a too early issued prefetch.

Compiler algorithm for cache injection includes support for inserting **`lprefetch`** and **`write_back`** instructions. Compiler support at the consumer side is similar to one used in the classical prefetching, except that it should be applied only to write-shared data. At the producer side, the compiler should only replace last store instruction to a memory block by **`write_back`** instruction. Hence, compiler support is less sophisticated than in data forwarding, since there is no need for identification of future consumers.

Hardware support for cache injection, in addition to support needed for prefetching and forwarding, includes the *injection table* with capability of snooping, and **`lprefetch`** instruction to fill the *injection table*. Cache prefetching requires prefetch instructions, lock-up free caches, and a buffer that keeps pending prefetches. Data forwarding requires a **`forward`** instruction, lock-up free caches, a deeper write buffer for pending forwards, and the ability for a cache to accept data that it has not requested. Additional complexity for cache injection is not large compared to overall complexity for both prefetching and forwarding.

The proposed solution can evolve by extending the function of the *injection table*. In the basic solution, an entry of the *injection table* contains the address of the requested cache line. If write-shared data demonstrate strong spatial locality and producer-consumer sharing pattern, an entry in the *injection table* can contain both the first and the last address of a memory block that is shared, defining a window in the address space. In this case, during write-back bus transactions, each processor checks its *injection table* to see if the bus address belongs to the address range defined in some of the entries. If it is the case, the data will be caught and stored into the cache.

`open_window Laddr, Haddr`
This instruction opens an address window in the injection table. **`Laddr`** *and* **`Haddr`** *represent the first and the last address of the window, respectively.*

`close_window Laddr`
This instruction checks the injection table, and if there is an open window which starts at the address **`Laddr`***, it closes that window.*

Figure 1. Proposed instructions.

Compiler algorithm, at the consumer(s) side, should find such address windows and insert corresponding

instructions to open (**`open_window Laddr, Haddr`**) and close (**`close_window Laddr`**) an address window. **Figure 1** contains the description of the proposed instructions. At the producer side, the compiler algorithm is the same as in the basic solution. This solution requires the *injection table* to be doubled in size (each entry contains two address fields) and simple combinational logic to compare addresses. The organization of the *injection table* is shown in Figure 2.

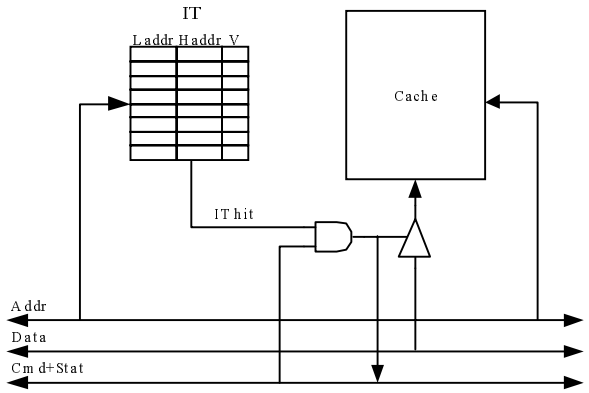

Figure 2. Organization of the *injection table*. **Description**: Each entry of the *injection table* table includes two address fields to define an address window and the *valid* bit. The **`open_window`** instruction defines the address fields (Laddr, Haddr) and sets the *valid* bit. The **`close_window`** instruction resets the *valid* bit.

4.2. An Example

To make our approach more concrete, an example code is considered (**Figure 3**). Matrix **A** is partitioned in such way that each processor modifies its own row, based on the value **`myVal`**, which is computed over the entire matrix **A**. This computation is repeated over several timesteps. Barriers are used to synchronize processors between computing of their copy of **`myVal`** (it depends on **`MyID`**) and using it to modify their own row.

A simple performance analysis, aimed to represent the advantages of the proposed solution compared with the existing ones, is performed. The CPU read misses and overall bus traffic are measured. Processor stall time waiting for necessary data from memory, directly depends on CPU read misses. To make the analysis simpler the following assumptions are adopted: cache line size is 16B (2 elements of matrix **A**), the number of iterations needed to prefetch ahead is 6, and the cache size is large enough to accept all elements of matrix **A**. A bus-based multiprocessor with the MESI write-back invalidation cache coherence protocol and the release memory consistency model is assumed. The addresses are 32 bits, and the bus commands are 8 bits. During the analysis only references to matrix **A** are considered. Base code shown in **Figure 3** (Base) is extended to support cache prefetching (referred to as Pref), data forwarding (Forw), and cache injection (Inject).

Base. During the first iteration over the *t* loop, all processors fetch elements of matrix **A**. Therefore, at the first barrier each processor has all elements in its local cache in *Shared* state. In the second phase, each processor modifies its own row, thus initiating invalidate operations. At the second barrier, each processor is the exclusive owner of its own row, and all other elements are invalid. Consequently, in the next iteration each processor fetches all elements of matrix **A**, except its own row, which is already in its cache. Number of CPU read misses during the execution of the base program is given below:

$N_{CPU-RM} = [NumProcs + (t\_max - 1) \cdot (NumProcs - 1)] \cdot M / 2$.

The traffic is split into data and address+command bus traffic. Data and address+command bus traffic per processor are:

$Traffic_{Data} = N_{CPU-RM} \cdot 16$, and

$Traffic_{AddrCmd} = (N_{CPU-RM} + N_{INV}) \cdot 5$, respectively; N_{INV} represents the number of bus invalidations, $N_{INV} = M \cdot t\_max / 2$.

```
/* NumProcs = total numberof processors */
/* MyID = this processor's ID number */
/* shared matrix A; each processor owns a row */

shared double A[NumProcs][M];
for(t=0; i<t_max; t++) {
   local double myVal =0.0
   for(p=0; p<NumProcs; p++) {
     for(i=0; i<M; i++)
       myVal+=foo(A[p][i], MyID];
   }
   barrier(B, NumProcs);
   for(i=0; i<M; i++)
     A[MyID][i]=goo(A[MyID][i],myVal);
   barrier(B, NumProcs);
}
```

Figure 3. An example representing the producer-consumer data sharing scenario (Base).

Pref. The code modified to support cache prefetching is shown in **Figure 4**. Software pipelining transformation of the innermost loop is necessary to issue prefetches enough iterations ahead (6 iterations). In this case, we can approximately expect that all CPU read misses will be eliminated. However, the overall traffic is the same as during the execution of the code in **Figure 3**.

Forw. Base example modified to support data forwarding is shown in **Figure 5**. Each processor, after the producing of a cache block is finished, forwards the cache block to the future consumers (in this case, to all

other processors). In that way, all processors will find the requested data in their caches in the next outermost loop. It should be noted that data forwarding cannot eliminate read misses in the first iteration over *t*. Number of CPU read misses during the execution of the Forw example is $N_{CPU-RM} = NumProcs \cdot M/2$. However, this approach does not reduce the overall bus traffic. The traffic is the same as in the previous cases if we modify the code to skip useless forwarding instructions in the last iteration of the outermost loop *t*=*t_max*-1.

```
shared double A[NumProcs][M];

for(t=0; i<t_max; t++) {
   local double myVal =0.0
   for(p=0; p<NumProcs; p++) {
   for(i=0; i<6; i+=2)
     prefetch(&A[p][i]);
   for(i=0; i<M-6; i+=2) {
     prefetch(&A[p][i+6]);
     myVal+=foo(A[p][i], MyID];
     myVal+=foo(A[p][i+1], MyID];
   }
   for(i=M-6; i<M; i+=2) {
     myVal+=foo(A[p][i], MyID];
     myVal+=foo(A[p][i+1], MyID];
   }
   barrier(B, NumProcs);
   for(i=0; i<M; i++)
     A[MyID][i]=goo(A[MyID][i],myVal);
  barrier(B, NumProcs);
}
```

Figure 4. Example with prefetching (Pref).

```
shared double A[NumProcs][M];
for(t=0; i<t_max; t++) {
   local double myVal =0.0
   for(p=0; p<NumProcs; p++) {
      for(i=0; i<M; i++)
         myVal+=foo(A[p][i], MyID];
   }
   barrier(B, NumProcs);
   for(i=0; i<M; i+=2) {
      A[MyID][i]=goo(A[MyID][i],myVal);
      A[MyID][i+1]=goo(A[MyID][i+1],myVal);
      for(int j=0; j<NumProcs; j++)
          forward(&A[MyID][i], j);
   }
   barrier(B, NumProcs);
}
```

Figure 5. Example with data forwarding (Forw).

Inject. Example code with cache injection is presented in **Figure 6**. Each processor, as a consumer, opens an address window between **`&A[0][0]`** and **`&A[NumProcs-1][M-1]`** by executing **`open_window`** instruction. After the data producing is finished, each processor initiate a write-back bus cycle by executing **`write_back`** instruction. During the write-back bus transaction, each processor inspects its *injection table* to see if the current bus address belongs to one of the defined address windows. If it is the case, the processor catches the data from the data bus and puts it into its cache. In this way, all potential consumers fetch the data during the same write-back bus transaction. After the execution of the outermost loop is over, each processor closes the address window by executing **`close_window`** instruction. Number of CPU read misses during the execution of the Inject example is $N_{CPU-RM} = NumProcs \cdot M/2$. The number of CPU read misses can be further reduced by using the following mechanism. A barrier is inserted after **`open_window`** instruction to force synchronization of all processors; also, the injection mechanism is allowed during the read bus transactions. Under this assumptions, all (*NumProcs*-1) processors can fetch a cache block during the read bus transaction caused by read miss seen by a processor, the first reader of that cache block. In that way, each processor sees $N_{CPU-RM} = M/2$ CPU read misses, on average. However, unlike previous examples, this approach reduces the overall bus traffic. Data and address+command bus traffic per processor are:
$Traffic_{Data} = (N_{CPU-RM} + N_{WB}) \cdot 16$, and
$Traffic_{AddrCmd} = (N_{CPU-RM} + N_{INV} + N_{WB}) \cdot 5$, respectively;
$N_{WB} = M \cdot t\_max/2$ represents the number of the write-back bus transactions; $N_{INV} = M \cdot t\_max/2$. The number of write-back bus transactions can be also reduced if we skip useless **`write_back`** instructions in the last iteration over *t* loop, $N_{WB} = M \cdot (t\_max - 1)/2$.

```
shared double A[NumProcs][M];

open_window(&A[0][0], &A[NumProcs-1][M-1]);
for(t=0; i<t_max; t++) {
   local double myVal =0.0
   for(p=0; p<NumProcs; p++) {
      for(i=0; i<M; i++)
         myVal+=foo(A[p][i], MyID];
   }
   barrier(B, NumProcs);
   for(i=0; i<M; i+=2) {
      A[MyID][i]=goo(A[MyID][i],myVal);
      A[MyID][i+1]=goo(A[MyID][i+1],myVal);
      write_back(&A[MyID][i]);
   }
   barrier(B, NumProcs);
}
close_window(&A[0][0], &A[NumProcs-1][M-1]);
```

Figure 6. Example with cache injection (Inject).

Table 1 shows the number of CPU read misses, the data and address+command bus traffic per a processor during the execution of all presented examples. We assume that *NumProcs*=16, *t_max*=10, and *M*=100. This small example shows that the cache injection significantly reduces bus traffic for applications with producer-consumer sharing pattern.

This approach is also very effective for synchronization variables such as locks, barriers, etc. For each synchronization variable each processor should open the window in the *injection table*, and force the write-back bus transaction by executing a **`write_back`** instruction, after the last modification of the variable.

Table 1. Comparison of performance for Base, Pref, Forw, and Inject.

	Base	Pref	Forw	Inject
$N_{CPU\text{-}RM}$	7550	≈0	800	50
$Traffic_{Data}$ [x10^3B]	120	120	120	8
$Traffic_{AddrCmd}$ [x10^3B]	40	40	40	5

Compiler support needed for cache injection can be even simpler compared to prefetching and forwarding. This approach, also, results in lower instruction overhead.

5. Experimental Methodology

We evaluate the performance of cache injection comparing the base system with the MESI write-back invalidation protocol, and the base systems extended with prefetching and cache injection, referred to as Base, Pref, and Inject, respectively.

The simulation is done using Limes tool [10] - a tool for execution-driven simulation of shared memory multiprocessors. In our experiments, we use several kernels well suited to demonstrate various data sharing patterns, and parallel applications from the SPLASH-2 application suite. Simple heuristics based on application behavior are used to insert classical prefetch and newly proposed instructions.

The initial performance evaluation, aimed to explore the potential of cache injection, shows that cache injection could significantly improve the performance relative to the base system, both with and without prefetching. These performance benefits are accomplished with lower bus traffic and reduced miss rate, especially for write-shared data. Performance improvements for some kernel benchmarks which exhibit producer-consumer sharing pattern are quite significant (speedups of over 60%).

6. Conclusions

This paper presents a novel software-controlled technique for tolerating memory latency in bus-based shared memory multiprocessors. This technique, called 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, in conditions typical of bus-based architectures.

Preliminary performance evaluation shows that cache injection provides significant improvements in tolerating the memory latency for irregular communication, synchronization, and producer-consumer sharing patterns. Detailed in-progress simulation-based performance evaluation concentrates on the impact of cache injection on execution time in state-of-the-art bus-based multiprocessors, both when applied alone and in combination with cache prefetching and data forwarding.

References

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

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

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

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

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

[6] P. Trancoso, J. Torrellas, "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.

[7] J. Skeppstedt, P. Stenstrom, "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.

[8] V. Milutinovic, A. Milenkovic, G. Sheaffer, "The Cache Injection/Cofetch Architecture: Initial Performance Evaluation," *Proceedings of the 5th MASCOTS*, IEEE Computer Society Press, January 1997.

[9] A. Milenkovic, V. Milutinovic, "Lazy Prefetching," *Proceeding of the 31st HICSS*, IEEE Computer Society Press, Vol. 7, January 1998.

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