The Impact of Speeding up Critical Sections with Data Prefetching and Forwarding[1]

Pedro Trancoso and **Josep Torrellas**

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign, IL 61801
Email: trancoso,torrella@csrd.uiuc.edu

Abstract

While shared-memory multiprocessing offers a simple model for process synchronization, actual synchronization may be expensive. Indeed, processors may have to wait for a long time to acquire the lock of a critical section. In addition, a processor may have to stall for a long time waiting for all of its pending accesses to complete before releasing the lock. To address this problem, we target well-known optimization techniques to specifically speed-up accesses to critical sections. We reduce the time taken by critical sections by applying data prefetching and forwarding to minimize the number of misses inside these sections. In addition, we prefetch and forward data in exclusive mode to reduce the stall time before lock release. Our evaluation shows that a simple prefetching algorithm is able to speed-up parallel applications significantly at a very low cost. With this optimization, five Splash applications run 20% faster on average, while one of them runs 52% faster. We also conclude that more complicated, forward-based optimizations are not justified.

1 Introduction

While shared-memory multiprocessing offers a simple model for process synchronization, actual synchronization is often an expensive task. In particular, a common form of synchronization, namely critical sections, is often a source of large overheads in parallel programs. Indeed, to enter the critical section, a processor may have to wait for a long time before it can acquire the lock. In addition, before a processor can release the lock and exit the critical section, it may have to stall for a long time waiting for all of its pending requests to complete. These two overheads often account for a significant fraction of the execution time in parallel programs. Clearly, given that these overheads are concentrated in a very small section of the code, there should be an easy way to determine if they can be optimized away.

One possible approach to speed up critical sections is to provide hardware support to make the acquire and release operations very fast [3]. This approach, however, may not be the most cost-effective one. This is because a large fraction of the execution time of a critical section is often consumed by the ordinary accesses in the section as opposed to the acquire and release accesses. This occurs, for example, when the ordinary accesses suffer many misses. In fact, misses in the ordinary accesses of the critical section are likely: for instance, 40-85% of the total data read misses in the second-level cache for the five Splash applications [11] considered in this paper occur in critical sections.

A more promising approach to speed up critical sections is to cluster the lock with the data that the lock protects. Then, when a processor acquires the lock, the protected data is made automatically available to the processor. This general approach was used by the KSR-1 designers [5] and by Kagi *et al* [4]. One difference between these two works is the granularity, which is sub-page (128 bytes) for the KSR-1 and cache line in the second case. Both works, however, have the problem that the variables associated with the lock have to fit in the space available and have to be statically allocated into that same space.

It is better, instead, to speed up critical sections with a technique that is cheaper or that has the flexibility of being useful for other purposes as well. For this reason, we choose data prefetching and forwarding. Data prefetching involves loading the data into the cache ahead of the time of its use [1, 9, 13]. Forwarding involves sending the data to the cache of the future consumer so that by the time the consumer needs the data it can access it locally from its cache [6, 7, 10]. While these techniques have been studied by many researchers, our contribution in this paper is to apply them efficiently and systematically to critical sections. With such a targeted application to key sections of the code, we hope to simplify the compiler analysis required while at the same time provide a multiplicative effect on the rest of the program.

The specific approach that we take is to reduce the time taken by critical sections by using data prefetching and forwarding to minimize the number of misses inside critical sections. In addition, we prefetch and forward data in exclusive mode to reduce the stall time before lock releases. Overall, we conclude that simple prefetching algorithms are enough. With such algorithms, five Splash applications run 20% faster on average, while one of them runs 52% faster. More complicated, forward-based algorithms are not as cost-effective.

The rest of this paper is organized as follows: Section 2 describes the optimizations and the algorithms that implement them; Section 3 presents the experimental setup; Section 4 evaluates the optimizations; finally, Section 5 concludes.

2 Optimizing Critical Sections

The goal of the optimizations presented in this paper is to reduce the execution time of critical sections, namely the pieces of code surrounded by a lock acquire and a release. What motivates the interest in this issue is that many

[1]This work was supported in part by the National Science Foundation under grants NSF Young Investigator Award MIP 94-57436 and RIA MIP 93-08098, ARPA Contract No. DABT63-95-C-0097, NASA Contract No. NAG-1-613, Intel Corporation, and the Portuguese government under scholarships JNICT FMRH/BD/1009/94 and JNICT PRAXIS XXI/BD/5877/95.

Splash [11] and other parallel applications have a significant synchronization overhead. For example, for the five Splash applications used in this study, we measured an average synchronization time of 24% of the total execution time (Section 4.4). Speeding up critical sections may therefore have a large performance impact. In addition, since critical sections are fairly small pieces of code, it is conceivable that a detailed analysis may give clues on how to optimize them.

In general, the time taken by critical sections comes from four major sources: (1) instruction execution, (2) processor stall to enforce that only one process is inside the critical section at a time, (3) processor stall waiting for necessary data from memory, and (4) processor stall to enforce the memory consistency model. In this paper, we focus on eliminating the latter two for a release consistency memory system. Indeed, we attempt to decrease the number of accesses to main memory seen by the processor to only one per critical section and eventually to zero. Furthermore, we try to hide stalls required to enforce the memory consistency model.

To do so, we use the well-known techniques of data prefetching and forwarding. Data prefetching involves loading the data into the cache ahead of the time of its use so that by that time the data is already available and no read stall occurs. Forwarding involves the producer processor sending the data to the cache of the future consumer so that by the time the consumer needs the data it can access it locally from its cache. In the following sections, we first describe the optimizations, then present the algorithms, and finally discuss the required architectural support. A more detailed discussion can be found in [12].

2.1 Description of the Optimizations

The first optimization attempts to reduce to one the number of accesses to main memory seen by the processor in the critical section. This is done by prefetching right after the acquire all the shared variables that are read in the critical section. Since the prefetches access the memory in a pipelined manner, only one memory access latency will be seen. Even this latency, of course, could be removed if the prefetches were moved outside the critical section. In that case, however, we would run the risk of getting invalidation messages for the prefetched data before acquiring the lock. Finally, we note that some critical sections access data by referencing a pointer. In such cases, it is not possible to overlap the prefetching of the pointer and the prefetching of the data pointed to by the pointer. To minimize stall time, the prefetches must be scheduled carefully. The optimization can be expressed as follows:

Optimization 1 (Pref) *Whenever a lock is acquired, prefetch all the shared data that will be read within the critical section.*

The second optimization attempts to hide the single memory access seen by the processor in the first optimization. Indeed, the data to be communicated is forwarded from the first processor to the second right before the first processor releases the lock. These forward statements cost a few cycles to issue. In addition, the issuing processor has to receive an acknowledgement making the completion of each forward before the processor can issue the release. Fortunately, forwards are not blocking and, therefore, can be overlapped with other forwards or with computation. In the best case, when the second processor acquires the lock, all the data that it needs in its critical section is already in its cache.

There are some problems with this use of data forwarding. The first one is that the processor issuing the forwards needs to know which other processor to send the data to. The most obvious way of doing this is to keep in memory a queue of all the processors waiting for that lock. Then, the releasing processor can read this queue to determine what processor to forward the data to. Reading such queue, of course, involves some overhead. We note in passing that this queue can be used to forward the lock variable itself too.

The second problem is that this optimization works only if, at the time that the lock is about to be released, there is a processor waiting to acquire it. To handle the case where no processor is waiting, we have to combine this optimization with the *Pref* optimization discussed above: before issuing the forwards, the processor has to check if there is any processor waiting. If none is waiting, the forwards are skipped. The consumer processor, after acquiring the locks will first attempt to prefetch data right after the acquire. If the data has already been forwarded to its cache, the prefetch requests will be terminated in the cache in one cycle.

Finally, it is possible that a given lock variable is used in two different critical sections that perform different operations. In this case, the forwards that we suggest may end up sending data that is useless to the acquiring processor. The approach that we follow is to forward the union of variables read by the two critical sections. Overall, this optimization can be expressed as follows:

Optimization 2 (Forw) *Whenever a lock is acquired, prefetch all the shared data that will be read within the critical section. Right before executing the release operation, if there is a processor waiting to acquire the lock, forward that same data to the waiting processor.*

The two optimizations presented focus on reads only because we assume that, while read misses stall the processor, writes do not. However, both writes and reads can cause the processor to stall to enforce the memory consistency model. For example, a release may not be issued because the processor is waiting for invalidation acknowledgements resulting from a write. A known way to reduce the effect of this problem is to originally request the data with the intention to write. This means requesting the data in state exclusive at the time of the read. When the data is read in, all sharers are invalidated; when the data is finally written, there is no need to send invalidations. This idea can be combined with prefetching (which is then called exclusive prefetching [9]) or with forwarding (which we call exclusive forwarding). Therefore, we have two more optimizations:

Optimization 3 (ExPref) *Whenever a lock is acquired, prefetch in exclusive mode all the shared data that will be written within the critical section, and prefetch all the remaining shared data that will be read within critical section.*

Optimization 4 (ExForw) *Whenever a lock is acquired, prefetch in exclusive mode all the shared data that will be written within the critical section, and prefetch all the remaining shared data that will be read within the critical section. Right before executing the release operation, if there is a processor waiting to acquire the lock, forward to* *the processor in exclusive mode all the written shared data and forward to the processor the rest of the read shared data.*

These optimizations perform better for small- to medium-sized critical sections and when locks suffer high contention. For large-sized critical sections, not all the data may be accessed. These characteristics are typical of critical sections that protect control data structures rather than application data. For application data, the critical sections are usually large and their code accesses a large set of data.

2.2 Algorithms

Figure 1 shows the pseudo-code for the algorithms that implement the optimizations presented before. We start first with the *Pref* algorithm. We see that the variables that are read in the critical section are grouped in the *READ* list. The union operator that we use is a special operator that performs the union of the memory lines. In particular, for each variable that is read, we try to calculate its memory line. Then, we add the variable to the *READ* list only if no variable already in the *READ* list is mapped to the same line or if the address of the new variable cannot be determined at compile time.

In the algorithm, the variables inserted in the *READ* list depend on how aggressive the algorithm is. Indeed, since the critical section may have several if statements, we have different alternatives. For example, we may choose to insert only the variables that are read no matter what the path taken by the code is. In such case, no useless data will be prefetched. Alternatively, we can insert all variables read in any path. For the experiments in Section 4 we use both schemes. For all applications except *Pthor*, we insert all variables read in any path because the critical sections are very small. For *Pthor*, instead, we insert only variables read no matter what the path is. This is because one critical section in *Pthor* is very large. Finally, we note that, in the algorithm, the order in which the variables are inserted in the *READ* list is the same as the one in which they are taken out and the corresponding prefetches placed in the code. The prefetches are therefore placed in the order of the reads in the code.

The *Forw* algorithm starts by computing the *READ* list for each acquire and inserting the prefetches like in the *Pref* algorithm. Then, it computes the union of the *READ* lists for all the acquires of the same lock. This new list, called *TOTAL_READ*, contains the variables to forward at the end of every critical section protected by that lock. The order of the variables in the list determines the order in which the forward instructions should be inserted. Note that this aggressive approach may have some problems. First, *TOTAL_READ* may include variables that are not present in the cache at the time the forward instruction is issued. This situation can be handled by either transforming the forward into a "no-op" or by requesting the data from the memory system and then forwarding it. We chose to implement the first solution for simplicity reasons. The second problem is that some of the forwarded variables may be useless to the destination processor. We could obviously take the more conservative approach of generating *TOTAL_READ* as the intersection of all the *READ* lists. However, the simple critical sections of our applications do not justify doing so.

The *Forw* algorithm needs the ID of the processor to which the data should be sent. This *Dest* ID is determined by issuing a non-blocking read to the lock queue to find

Pref Algorithm:

```
for each critical section C
  READ(C) = ∅
  for each statement s in C from top to bottom
    if (s includes a read to variable v)
      READ(C) = READ(C) ∪ v
  for each variable v in READ(C)
    create ``Prefetch(v)'' statement
    insert it in C right before any non-prefetch
     statement
```

Forw Algorithm:

```
perform the Pref algorithm saving READ(C) for
 every critical section C
for each lock variable L
  TOTAL_READ(L) = ∅
  for each C that uses lock variable L
    TOTAL_READ(L) = TOTAL_READ(L) ∪ READ(C)
  for each C that uses lock variable L
    for each variable v in TOTAL_READ(L)
      create a ``Forward(v, Dest)'' statement
      insert it in C right before the releases
```

ExPref Algorithm:

```
for each critical section C
  ACCESS(C) = ∅
  for each statement s in C from top to bottom
    if (s includes a read to variable v)
      ACCESS(C) = ACCESS(C) ∪ v
    if (s includes a write to variable v)
      ACCESS(C) = ACCESS(C) ∪ SetExclusive(v)
  for each variable v in ACCESS(C)
    if (IsExclusive(v))
      create a ``Prefetch_ex(v)'' statement
    else
      create a ``Prefetch(v)'' statement
    insert it in C right before any non-prefetch
     statement
```

ExForw Algorithm:

```
perform the ExPref algorithm saving ACCESS(C) for
 every critical section C
for each lock variable L
  TOTAL_ACCESS(L) = ∅
  for each C that uses lock variable L
    TOTAL_ACCESS(L) = TOTAL_ACCESS(L) ∪ ACCESS(C)
  for each C that uses lock variable L
    for each variable v in TOTAL_ACCESS(L)
      if (IsExclusive(v))
        create ``Forward_ex(v, Dest)'' statement
      else
        create ``Forward(v, Dest)'' statement
      insert it in C right before the releases
```

Figure 1: *Pref*, *Forw*, *ExPref*, and *ExForw* algorithms. *Dest* refers to the processor that is waiting to acquire the lock next. To simplify the forwarding algorithms, we do not show that the releasing processor reads the queue of waiting processors to determine what processor to forward the data to.

```
LOCK(l);        LOCK(l);             LOCK(l);
a = b + c;      Prefetch(b);         Prefetch(b);
d = f→g;        Prefetch_ex(a);      Prefetch_ex(a);
UNLOCK(l);      Prefetch(f);         Prefetch(f);
                Prefetch(f→g);       Prefetch(f→g);
                Prefetch_ex(d);      Prefetch_ex(d);
    (a)         a = b + c;           a = b + c;
                d = f→g;             d = f→g;
                UNLOCK(l);           Forward(b,Dest);
                                     Forward_ex(a,Dest);
                                     Forward(f,Dest);
                      (b)            Forward(f→g,Dest);
                                     Forward_ex(d,Dest);
                                     UNLOCK(l);

                                            (c)
```

Figure 2: Example of a critical section (Chart(a)) after applying the *ExPref* (Chart (b)), and *ExForw* (Chart (c)) algorithms. The code after applying *Pref* and *Forw* is exactly the same as shown without the *Prefetch_ex* and *Forward_ex* calls. Variables b and c are allocated in the same cache line.

out which processor is waiting next. This read, which is not shown in the algorithm, is issued right after the acquire. If there is no processor waiting in the lock queue, then the processor can send the forwards to memory or can skip the forwards altogether. In the experiments of Section 4, this case is handled by the processor forwarding to itself, effectively performing "no-op" operations.

The *ExPref* and *ExForw* algorithms are based on the *Pref* and *Forw* algorithms. However, we use the *ACCESS* and *TOTAL_ACCESS* lists instead of the *READ* and *TOTAL_READ* lists respectively. The former two lists contain variables that are read or written, while the latter two contain only variables that are read. In *ACCESS* and *TOTAL_ACCESS*, the variables that are written are marked with the attribute *exclusive*, and the corresponding line is prefetched or forwarded in exclusive mode. Such marking is performed with the *SetExclusive* function and checked with the *IsExclusive* function. The *exclusive* state for a line overwrites the *non-exclusive* state.

A small example of the application of these algorithms is shown in Figure 2. In the figure, variables b and c are allocated in the same cache line. The algorithms presented are very simple and could be refined in many ways. For example, the forwards could be performed at the same time as the last writes to the variables, like it was proposed in [6, 10], thereby saving cycles. Similarly, the prefetch statements could be scheduled better, specially in the context of variables accessed via pointer dereferencing. Finally, our algorithms as they are now prefetch or forward only one iteration of a loop. For all these reasons, the results in Section 4 are conservative.

2.3 Architectural Support

The architectural support required by the prefetch operation is modest. A popular implementation of prefetching requires a prefetch instruction that loads data without blocking and takes no exceptions, lock-up free caches to handle multiple outstanding prefetches, and the buffering of some state for each pending prefetch until the prefetch transaction is completed.

The architectural support required by the forwarding operation defined here is a bit more complex. Indeed, there is the need to implement a per-lock queue where the processors waiting on the lock are inserted in FIFO order. This queue can be implemented in hardware [4] or software [8]. Then, we need to add a forwarding instruction, buffer some state for each pending forward until the forward transaction is completed, modify the cache coherence protocol to support forwarding transactions, and modify the caches to accept lines that have not been requested by the processor. Many optimizations to this base scheme are possible. For example, we could use Poulsen and Yew's [10] and Koufaty *et al*'s [6] combined *write-and-forward* instruction.

3 Experimental Setup

In this work we simulate a cache-coherent shared-memory machine (CC-NUMA) with 32 nodes. Each node includes a 4-Kbyte first-level write-through cache and a 64-Kbyte second-level write-back cache. Both caches have 4-word cache lines and are direct-mapped. Each of the nodes contains a part of the global memory and its directory. The processor on each node is a RISC processor with single instruction issue per clock cycle. A hit in the first-level cache is solved without penalty, while a first-level cache line fill takes 14 cycles if the line comes from the secondary cache and 50 cycles if the line comes from the local memory.

The interconnection network is modeled by a fixed latency value of 30 cycles for transmit time and 34 cycles for data transmission (both values for a one-way trip). Contention is modeled in the various resources of the architecture except in the network.

Table 1: Applications used in the experiments.

Apps.	Description	Problem Size
Barnes	Simulates the evolution of galaxies	1000 bodies 3 time steps
Cholesky	Cholesky factorization of a sparse matrix	bcsstk14.O 1806 columns
Mp3d	Simulates a rarefied hypersonic flow	10000 particles 10 time steps
Pthor	Simulates a digital circuit at the logic level	small RISC processor
Water	Simulates evolution of set of water molecules	343 molecules 2 time steps

This architecture is simulated on a simulator based on TangoLite [2], an execution-driven simulator. For this study, the memory simulator was enhanced with some extra functionalities like monitoring all the synchronization variables and introducing prefetching and forwarding directives.

The workloads run are some of the Splash [11] applications. A brief description of the applications and the problem sizes used are presented in Table 1. We run the applications with 32 processors. In these applications, we insert the extra prefetch and forward instructions "by hand".

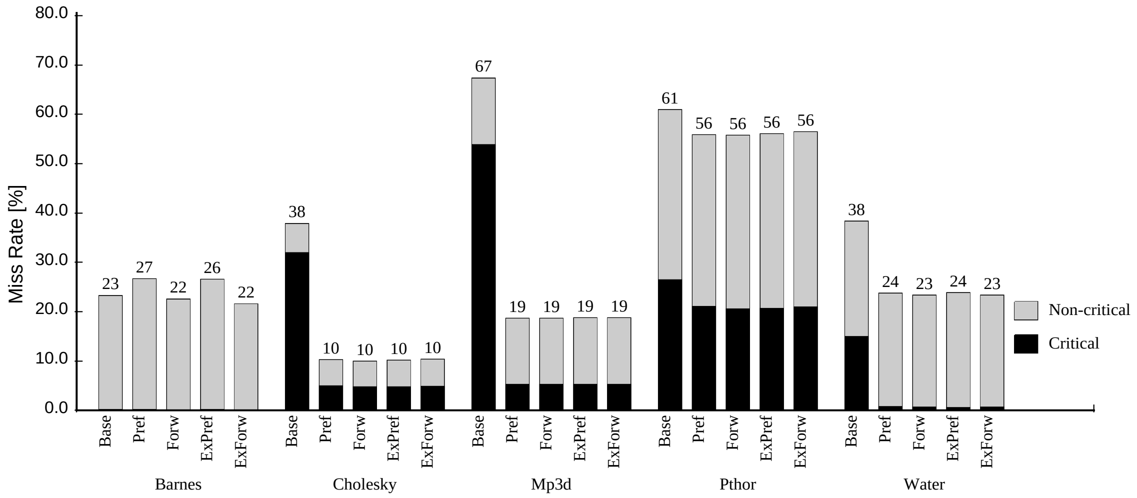

Figure 3: Read miss rate of the second-level cache for different levels of optimization.

4 Evaluation

In this section we evaluate the algorithms described in Section 2. Since some of the applications have specific problems that prevented scalability, we start by making some changes to some applications. Then, we present the impact of the proposed optimizations on the read miss rate, synchronization time, and execution time.

4.1 Some Problems in Applications

Before we apply the algorithms presented in Section 2, we make changes to some of the applications to reduce the time processors waste waiting in barriers. Much of this time is due to load imbalance in the application. To be fair, this load imbalance needs to be addressed first, before applying the algorithms, since it is the most obvious synchronization problem.

In *Barnes*, the critical sections are very large. For example, some of them include several vector operations implemented using loops. Such large critical sections cause an excessive serialization and, consequently, load imbalance in the application. To fix the problem, we divide the critical sections into smaller critical sections. As a result, more operations can be done in parallel or in pipeline. This modification reduces the execution time by 2%.

Cholesky has load imbalance due to the different sizes of the matrix columns. To optimize the execution of the algorithm, the application groups the columns into *supernodes*. Because *supernodes* have different sizes, the work given to each processor is very different. To reduce this problem, we impose a tighter limitation on the *supernodes*' maximum size. As a result, their size ranges are smaller and therefore the work is more balanced. This optimization results in a 10% reduction in the execution time.

4.2 Impact of the Optimizations on Read Miss Rates

To determine the effect of our algorithms, we start by examining their impact on the read miss rate. We break down the read miss rate into two distinct categories, the miss rate due to the read misses within critical sections (*Critical*) and the rest (*Non-critical*). This breakdown allows us to see the weight of the critical sections in terms of the read miss rate. Then, we apply our algorithms and determine the new miss rate breakdown. We can then see how effective our optimizations are.

Figure 3 shows the read miss rate breakdown for the original and optimized codes. The figure shows the miss rate of the 64-Kbyte second-level cache. The miss rates are high because the first-level cache filters many of the references. In this figure, we divide the read miss rate into the *Critical* and *Non-critical* categories. For each application we show five bars which, from left to right, are: *Base* for the original code, *Pref* for the code with the prefetching optimization, *Forw* for the code with the forwarding optimization, *ExPref* for the code with the exclusive prefetching optimization, and *ExForw* for the code with the exclusive forwarding optimization.

From the figure, we can see that, with the exception of *Barnes*, the misses in the critical section contribute significantly to the miss rate of the *Base* version of the applications. Indeed, 40-85% of the miss rate comes from misses in the critical section. This suggests that, by targeting these small, well identified portions of code that are the critical sections, the overall miss rate can be reduced significantly.

If we now consider the other bars, we see that, in most cases, all optimizations have a similar effect. Such effect, however, varies depending upon the application. On the one hand, there are applications like *Cholesky*, *Mp3d*, and *Water* where the miss rate reduction is as large as 40-75%. On the other hand, for *Pthor*, the miss rate reduction is a modest 9%. In all these cases, of course, the miss rate reductions come from reducing the number of misses in the critical section. In *Barnes*, there are very few misses inside critical sections. However, inserting prefetches to eliminate these misses results in an increase in misses outside critical sections.

In these four applications, if we compare the bars corresponding to the different optimizations we observe that, against intuition, *Forw* does not do better than *Pref* and *ExForw* does not do better than *ExPref*. In Section 2, we stressed that forwarding optimizations can potentially

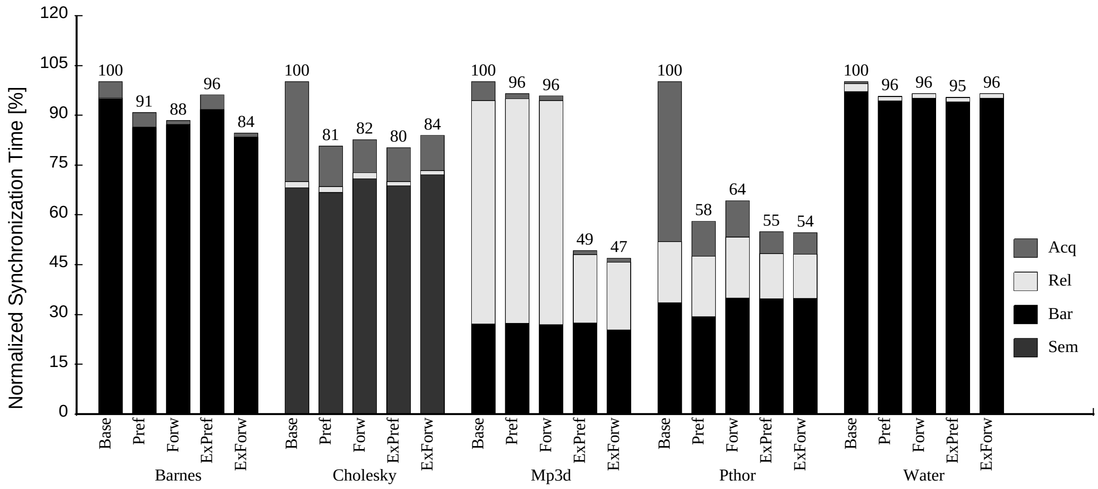

Figure 4: Impact of the optimizations on the synchronization time.

Table 2: Read miss rate coverage.

Application	Cholesky	Mp3d	Pthor	Water
Coverage	84.6	90.3	20.5	95.3

reduce the number of read misses even further than the prefetching optimizations. There are, however, two reasons for not observing much change in the miss rates. The first one is that cache misses that are only partially hidden by prefetching or forwarding are not shown in Figure 3. Consequently, while forwarding may partially hide some misses to a larger degree than prefetching, Figure 3 will not show it. The second reason for not observing much change in the miss rates is that, for some applications, sometimes there are no processors waiting to enter the critical section. In such cases, there is no difference between forwarding and prefetching. Overall, the results in Figure 3 already suggest that forwarding is unlikely to be much more effective than prefetching.

Figure 3 also shows the fraction of the misses in the critical section that have been removed with these optimizations. Such fraction we call the read miss rate *Coverage*. The exact coverage for the *Pref* optimization in the different applications is shown in Table 2. From the table we see that *Cholesky*, *Mp3d*, and *Water* have coverages larger than 80%. Obviously, the algorithms achieve their expected effect. In *Pthor*, instead, the coverage is only 21%. The reason for such a low coverage is that one of the critical sections in *Pthor* is very large. The code for this critical section is spread onto multiple files. As a result, our algorithm inserts many prefetches and forwards. Consequently, as indicated in Section 2.2 we used the conservative versions of the algorithms in Figure 1, which insert variables in the *READ* and *ACCESS* list only if they are read or accessed in all possible paths of execution within the critical section. This reduced the coverage. Finally, we excluded *Barnes* from Table 2 because its coverage does not have much meaning given that there are very few misses in the critical section to start with.

In conclusion, we found that, for the five applications, an average of 50% of the second-level cache read misses occur within critical sections. Our algorithms are able to eliminate on average over 58% of the misses in critical sections, thereby reducing the miss rate by about 35%.

4.3 Impact of the Optimizations on Synchronization Time

We now analyze the impact of our algorithms on the synchronization time. The time spent on synchronization can be divided into time waiting in barriers (*Bar*), time spent spinning on a lock before acquiring it (*Acq*), and time spent waiting for the completion of pending requests before releasing a lock (*Rel*). In addition, *Cholesky* has semaphores that arbitrate the assignment of work to threads in a centralized point. Semaphores include additional information like how many resources are available. The time that threads wait in these semaphores we call *Sem*, and is mostly due to load imbalance of the application. Consequently, our optimizations do not address the *Sem* time directly.

Figure 4 shows the breakdown of the synchronization time for the different applications and levels of optimization. For each application, we have five bars that are normalized to the *Base* time and are broken down into *Acq*, *Rel*, *Bar*, and *Sem* time. From the *Base* bars, we can see that there are two types of applications, namely those where *Bar* or *Sem* account for most of the time and those where *Acq* or *Rel* are important. The applications in the first group are *Barnes*, *Cholesky*, and *Water*, while the applications in the second group are *Mp3d* and *Pthor*. This distinction is important because our optimizations cannot target the first group of applications. The first group of applications need changes to balance the load among the processors.

The first two optimizations, namely *Pref* and *Forw*, attempt to reduce the *Acq* time. Indeed, as indicated in Section 2.1, they reduce the misses in critical sections, which speeds up the execution of critical sections and therefore reduces the time that other processors need to wait to get in. Such effect is observed in Figure 4: *Pref* eliminates a large fraction of the *Acq* time in *Cholesky*,

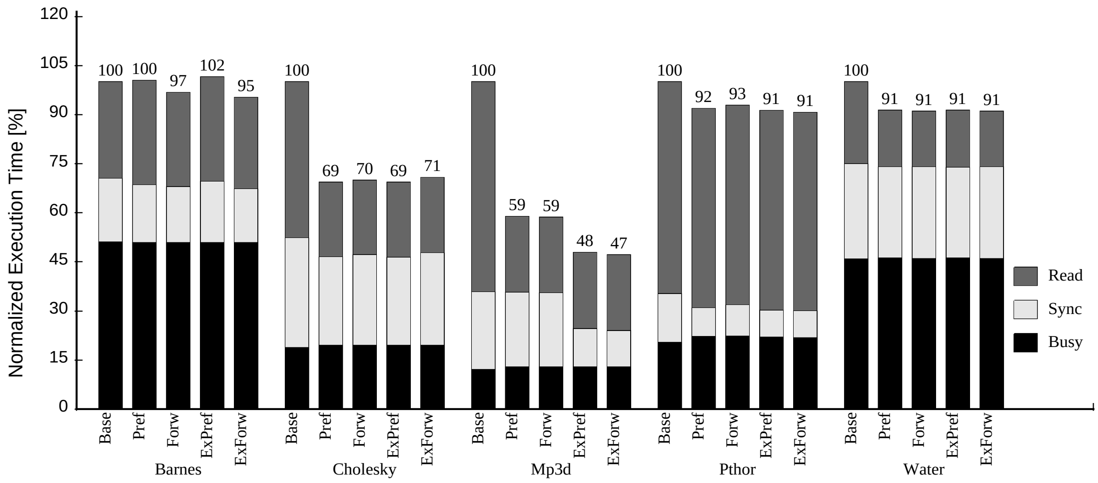

Figure 5: Impact of the optimizations on the execution time of the applications.

Mp3d, and *Pthor*, reducing the overall synchronization time significantly in two of these applications. Surprisingly, *Forw* does not do much better than *Pref*. It may be that the contention in the memory subsystem prevents forwarding from hiding the misses to a much larger extent than prefetching. In fact, *Forw* sometimes does worse than *Pref*. The reason is random changes in load balance in the application, which sometimes cause an increase in the *Bar* and *Sem* time. Overall, *Pref* and *Forw* reduce the synchronization time of the five applications by 16% and 15% on average respectively. After applying these optimizations, very little *Acq* time is left in the applications.

The next two optimizations, namely *ExPref* and *ExForw*, attempt to reduce the *Rel* time. This is because, as indicated in Section 2.1, data is prefetched and forwarded in state exclusive and, therefore, there is no need for a later network access that creates invalidations and acknowledgements that delay the release of the lock. These optimizations, therefore, will only have an impact on the applications with *Rel* time, namely *Mp3d* and *Pthor*. The reductions in *Rel* time are large, especially in *Mp3d*. Overall, *ExPref* decreases the synchronization time of *Pref* by 49% and 5% for *Mp3d* and *Pthor* respectively. The reason for the large improvement in *Mp3d* is that this application's critical sections are small and contain many write operations at the end of it. Finally, we note that, possibly for the reason explained before, both *ExPref* and *ExForw* have a similar impact.

The applications with large *Bar* or *Sem* time change relatively less because our optimizations do not target those times. Most of their changes are due to random variation in the *Bar* or *Sem* time due to load balance variations.

Overall, the impact of the algorithms on the synchronization time is a function of the contribution of the *Acq* and *Rel* time to the total synchronization time. For applications with large *Acq* and *Rel* time like *Mp3d* and *Pthor*, we get a large improvement of around 50%. For the other applications, we get more modest results. On average, after applying *ExForw*, the applications reduce their synchronization time by around 25%.

Table 3: Applications characteristics.

	Read Miss Reduction?	Sync Time Reduction?	Expected Improvement
Barnes	n	n	small
Cholesky	y	n	medium
Mp3d	y	y	large
Pthor	n	y	medium
Water	y	n	medium

4.4 Impact of the Optimizations on Execution Time

We now assess the impact of the optimizations on the execution time of the applications. Such impact will be a combination of the two effects presented so far: a reduction in read stall time resulting from eliminating some misses in the critical sections (Section 4.2) and a direct reduction in synchronization time (Section 4.3).

To gain insight into what applications may benefit from the optimizations, we summarize in Table 3 the findings in the previous two sections. The first column of the table lists the applications that showed a noticeable reduction in the miss rate (Section 4.2). These applications are *Cholesky*, *Mp3d* and *Water*. These applications, therefore, are likely to show reductions in the read stall time. Similarly, the second column of the table lists the applications that showed a noticeable reduction in the synchronization time (Section 4.3). These applications are *Mp3d* and *Pthor*. As a result, the data in the table suggests that *Barnes* is unlikely to benefit much from the optimizations; *Cholesky*, *Pthor*, and *Water* are likely to benefit to a certain extent; and finally, *Mp3d* is likely to benefit the most.

The actual measurements of the total execution time of the applications are shown in Figure 5. For each application, we show the *Base*, *Pref*, *Forw*, *ExPref*, and *ExForw* environments, all normalized to *Base*'s execution time. For each application, the bars are subdivided into three categories: computation time (*Busy*), synchronization time (*Sync*), and stall time due to read misses (*Read*).

As predicted in Table 3, the figure shows that *Mp3d* has the largest execution time reduction. Such reduction, which reaches 53% in the best case, comes from both a smaller *Read* and a smaller *Sync* time. Again, in agreement with Table 3, *Cholesky*, *Pthor*, and *Water* have medium to large execution time reductions. The reductions are 31%, 9%, and 9% for *Cholesky*, *Pthor*, and *Water* respectively. While the speedups in *Cholesky* and *Water* come from a lower *Read* time, the speedups in *Pthor* come from a lower *Sync* time. Finally, the execution time reduction for *Barnes* is small (5%), as predicted in Table 3.

If we compare the different optimizations, we can observe that, in most cases, *Pref* is enough to obtain most of the speedups. In one case, however, namely *Mp3d*, *ExPref* has a clear advantage over *Pref*. For this reason, we recommend the use of the *ExPref* optimization. We also note that, in general, *Forw* and *ExForw* do not seem to provide any major advantage over *Pref* and *ExPref*. The exception is *Barnes*, where the forward-based optimizations perform better than the prefetch-based ones. This effect, however, is small. Overall, therefore, considering the extra cost of the forward-based optimizations, we conclude that the forward-based optimizations are not worth supporting.

It is well-known that software prefetching has the disadvantage of adding extra instruction overhead to the execution. This does not seem to be a problem for the applications considered. Indeed, it is true that we can observe a small increase in the *Busy* time for *Cholesky*, *Mp3d* and *Pthor*. However, the overall impact of this effect is negligible.

To summarize, we conclude that, with the simple *ExPref* optimization we can obtain nice speedups in the applications considered. On average, these applications run 20% faster. The speedup is larger for applications where the *Read* time is large, many of the read misses occur in the critical section, the *Sync* time is significant, and the *Acq* and *Rel* parts of the *Sync* time are large. When these conditions are met we can obtain large execution time reductions like 52% for *Mp3d*.

5 Conclusions

In parallel applications, accesses to critical sections may involve large overheads. Indeed, processors may have to wait for a long time to acquire a lock. In addition, they may have to stall for a long time waiting for all pending requests to complete before releasing it. To address these problems, this paper has proposed and evaluated simple optimizations that speed up accesses in critical sections. Our approach is to reduce the time taken by critical sections by using data prefetching and forwarding to minimize the number of misses inside critical sections. In addition, we prefetch and forward data in exclusive mode to reduce the stall time before lock release.

We conclude that a simple prefetching algorithm that we have presented is able to significantly speed up parallel applications at a very low cost. The algorithm prefetches the variables accessed in the critical section. Those variables that are written in the critical section are prefetched in exclusive mode. With this optimization, five Splash applications run 20% faster on average, while one of them runs 52% faster. We also conclude that more complicated, forward-based optimizations are not justified.

Acknowledgments

We thank Zheng Zhang for all his help during the development of this work and especially for the simulation environment that was provided. We also thank the referees and the graduate students in the I-ACOMA group for their feedback. Pedro Trancoso is supported by a scholarship from Portugal. Josep Torrellas is supported in part by an NSF Young Investigator Award.

References

[1] J. L. Baer and T. F. Chen. An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty. In *Proceedings of Supercomputing'91*, pages 179–186, November 1991.

[2] S. Goldschmidt. Simulation of Multiprocessors: Accuracy and Performance. Ph.D. Thesis, Stanford University, June 1993.

[3] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient Synchronization Primitives for Large-scale Cache-coherent Multiprocessors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 64–73, April 1989.

[4] Alain Kagi, Nagi Aboulenein, Douglas C. Burger, and James R. Goodman. Techniques for Reducing Overheads of Shared-Memory Multiprocessing. In *Proceedings of the 1995 International Conference on Supercomputing*, pages 11–20, July 1995.

[5] Kendall Square Research. *KSR1 Technical Summary*. Waltham, MA, 1992.

[6] D. A. Koufaty, X. Chen, D. K. Poulsen, and J. Torrellas. Data Forwarding in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 1995 International Conference on Supercomputing*, pages 255–264, July 1995.

[7] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, pages 63–79, March 1992.

[8] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[9] Todd Mowry and Anoop Gupta. Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, pages 87–106, 1991.

[10] David K. Poulsen and Pen-Chung Yew. Data Prefetching and Data Forwarding in Shared Memory Multiprocessors. In *Proceedings of the 1994 International Conference on Parallel Processing*, volume II, pages 276–280, August 1994. Also available as CSRD tech report No. 1330.

[11] J. P. Singh, W. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-91-469, Computer Systems Laboratory, Stanford University, April 1991.

[12] Pedro P. M. Trancoso. Performance Optimization Based on Characterizing Synchronization. Master's thesis, University of Illinois at Urbana-Champaign, October 1995.

[13] Zheng Zhang and Josep Torrellas. Speeding up Irregular Applications in Shared-Memory Multiprocessors: Memory Binding and Group Prefetching. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 188–199, June 1995.