Busy-wait barrier synchronization using distributed counters with local sensor

Guansong Zhang, Francisco Martinez[⋆],
Arie Tal, Bob Blainey

IBM Toronto Lab
Toronto
ON, L6G 1C7, Canada

Abstract. Barrier synchronization is an important and performance critical primitive in many parallel programming models, including the popular OpenMP model. In this paper, we compare the performance of several software implementations of barrier synchronization and introduce a new implementation, *distributed counters with local sensor*, which considerably reduces overhead on POWER3 and POWER4 SMP systems. Through experiments with the EPCC OpenMP benchmark, we demonstrate a 79% reduction in overhead on a 32-way POWER4 system and an 87% reduction in overhead on a 16-way POWER3 system when comparing with a fetch-and-add implementation. Since these improvements are primarily attributed to reduced L2 and L3 cache misses, we expect the relative performance of our implementation to increase with the number of processors in an SMP and as memory latencies lengthen relative to cache latencies.

Keywords. barrier, synchronization, multiprocessor, distributed counter

1 Introduction

A barrier is a synchronization primitive used in parallel programming languages. The barrier point is a program position at which each thread of execution enters in parallel and waits to proceed until all threads have reached the same point. Figure 1 (a) illustrates a barrier synchronization of the execution of four threads. Barriers are necessary in many parallel programs to ensure data integrity between phases of a program which are to be executed using multiple threads.

Barriers are used in both shared and distributed memory programming models [1] [2] [3]. We restrict our attention in this paper to software barrier implementations for cache-coherent shared memory or symmetric multiprocessor (SMP) systems. Software barrier implementation for SMP systems can generally be done using hardware synchronization support such as fetch-and-add or test-and-set, by using busy-wait polling techniques or by using some combination of the two.

In the OpenMP shared memory parallel language specifications[4][5] , barrier synchronization plays a significant role. Parallel regions as well as all of the defined work-sharing constructs are terminated by an implicit barrier synchronization (which may,

[⋆] Ph.D. candidate, research student visiting from the Technical University of Catalonia (UPC), Barcelona, Spain

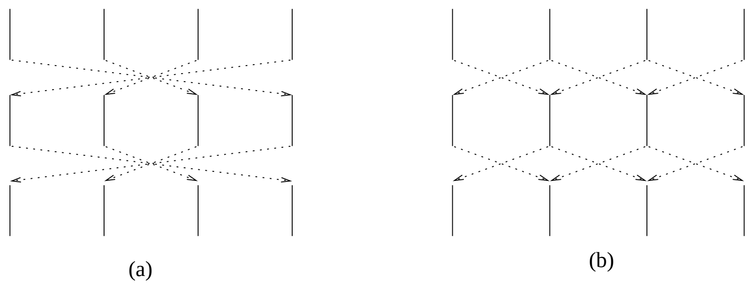

Fig. 1. Barriers

in some cases, be avoided using a NOWAIT clause). OpenMP also defines an explicit barrier directive defined to be used whenever necessary.

In this paper, we will study different ways of implementing busy-wait barrier synchronization on a multiprocessor system with shared memory, typically for parallel programming with OpenMP. We use the EPCC micro-benchmarks[6] to measure performance overhead of the different barrier implementations discussed here. All of the test data is based on the use of an explicit barrier, though the results can be applied equally well to implied barriers.

In section 2, we describe the POWER4-based system on which we performed our experiments and the source of some of the costs implicit in barrier synchronization. In section 3, we describe several implementations of busy-wait synchronization and finally describe our new approach, *distributed counters with local sensor*. In section 4, we show the results of timing and hardware performance monitoring experiments using the various barrier implementations. Finally, in section 5, we summarize our findings and describe future work.

2 Overhead of a barrier synchronization

There are many ways of implementing a barrier synchronization. In this paper, we focus on the so-called *centralized barrier*, simply because in a globally addressed memory system with wide bandwidth among node processors, such as a POWER4, the centralized barrier will make the coding much easier, and less error prone[1].

Several ways to implement this kind of barrier were suggested in [7] and re-examined on a ccNUMA system in [8]. Basically, a shared variable is updated as each thread arrives at the barrier point. The thread will leave the position if testing finds out that all the threads have arrived. To understand the overhead of the whole process, we first have a brief introduction on the hardware architecture used for testing, and our test environment.

[1] When there is a large number of processor nodes connected by a interconnection network with a specific topology, centralized barrier may not be preferable. While the barrier in Figure 1 (b) will be more interesting, where locality is emphasized.

2.1 POWER4 SMP architecture and software

The basic building block for a POWER4 SMP system can be found in [9]. It is a multi-chip module (MCM) with four POWER4 chips forming an 8-way SMP, as shown in Figure 2. Multiple MCMs can then be interconnected to form 16-, 24-, and 32-way SMPs.

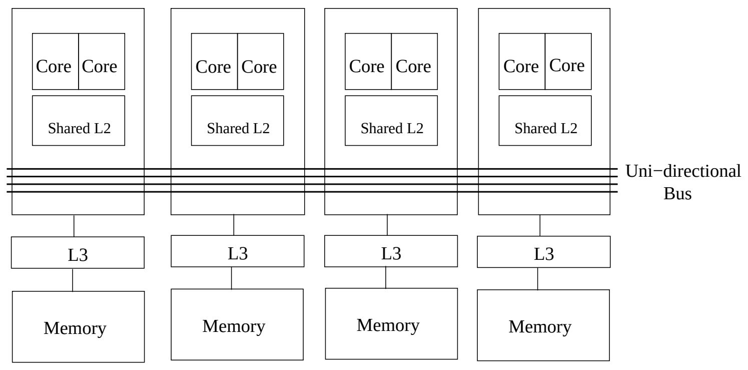

Fig. 2. POWER4 MCM structure

The logical interconnection of four POWER4 chips is point-to-point, with uni-directional buses connecting each pair of chips to form an 8-way SMP with an all-to-all interconnection topology. The fabric controller on each chip monitors (for example snoops) all buses and writes to its own bus, arbitrating between the L2 cache, I/O controller, and the L3 controller for the bus. Requests for data from an L3 cache are snooped by each fabric controller to determine if it has the data being requested in its L2 cache (in a suitable state), or in its L3 cache, or in the memory attached to its L3 cache. If any one of these is true, then that controller returns the requested data to the requesting chip on its bus. The fabric controller that generated the request then sees the response on that bus and accepts the data.

Up to four MCMs can be interconnected by extending each bus from each module to its neighboring module in one direction. Inter-module buses run at half the processor frequency and are 8-bytes wide. The inter-MCM topology is that of a ring in which requests and data move from one module to another module in one direction. As with the single MCM configuration, each chip always sends requests, commands and data on its own bus but snoops all buses for requests or commands from other MCMs.

The underlying software system is the SMP runtime library supporting the OpenMP standard. It is not a research system but production level software available in the IBM® XL Fortran and VisualAge C++® products. The runtime system implements schedule-based barrier synchronization as well as a fallback for the busy-wait method.

2.2 Testing benchmark

We use unmodified EPCC micro-benchmarks to test the performance of a barrier synchronization. As introduced in [10], the overhead here is considered as the difference between the parallel execution time and the ideal time, given by perfect scaling of the sequential program.

The parallel execution time is taken from the following code segment,

```
      dl = delaylength

      do k=0,outerreps
         start  = getclock()
!$OMP PARALLEL  PRIVATE(J)
         do j=1,innerreps
            call delay(dl)
!$OMP BARRIER
         end do
!$OMP END PARALLEL
         time(k) = (getclock() - start) *
     &              1.0e6 / dble (innerreps)
      end do
```

While the sequential reference time is measured through this code,

```
      dl = delaylength
      do k=0,outerreps
         start  = getclock()
         do j=1,innerreps
            call delay(dl)
         end do
         time(k) = (getclock() - start)*
     &              1.0e6 / dble (innerreps)
      end do
```

In the test program, the value of `outerreps` is set to 50, which is the default in the EPCC micro-benchmarks. The array variable `time` is used to compute the mean and standard deviation of the 50 measurements. Since we can exclusively access the machine, only the mean value is considered here.

2.3 Barrier overhead on an SMP system

The two main reasons for the overhead of a barrier synchronization are memory contention and traffic. We define *contention* as the effect produced by many threads accessing simultaneously the same data, and *traffic* as the amount of data per time unit moving through the bus.

Nevertheless, contention and traffic are not independent as some accesses to memory increase both (so, contention in one memory position is likely to increase traffic). Studying memory behavior through hardware counters can show us an approximation to both the contention and traffic that a concrete barrier implementation puts on the system.

In order to reduce the complexity of the analysis, we separate the barriers into phases:

– Signaling: Time to enter the barrier and notifying that we are in
– Leaving: Time to realize that the synchronization is over and we can leave the barrier

This classification can be useful because some implementations focus on reducing the impact of the first while others focus on the second.

3 Design of different barriers

In this section we summarize different designs for barrier implementation and their characteristics. Based on the discussion presented in the previous section, we try to reduce the overhead of a barrier by decreasing the contention for a shared memory location and reducing the volume of data transmitted on the network. We start with a simple barrier implementation using *Fetch-and-Add*.

3.1 Barrier with Fetch-and-Add

In this case, a global counter is allocated in the shared memory of a parallel system. Before a barrier starts, the counter will be set to the number of threads participating in the parallel region. As the threads come to the barrier point, each one will decrease the counter with an atomic fetch-and-add operation and then spin on checking the counter value until the result is zero.

This implementation is quite similar to the barrier implementation introduced in [11]. The difference is that instead of doing scheduling, a busy-wait method is enforced by letting each thread constantly read the shared counter. The whole process is shown in Figure 3.

In the diagram, we use square blocks to represent our data structure, and arrowed lines with comments to represent actions that a thread may apply to the structure, in this case, the shared memory counter.

On a POWER4 system, the fetch-and-add operation is implemented using a `lwarx` (load and reserve) and `stwcx` (store conditional) instruction pair. As shown in the following assembly code, the two instructions work together to conditionally store a word to an indexed memory position.

```
static inline int fetch_and_add (volatile int *mem, int val)
{
  int tmp, result;
  asm volatile("\n\
0:      lwarx   %0,0,%2     \n\
        add%I3  %1,%0,%3    \n\
        stwcx.  %1,0,%2     \n\
```

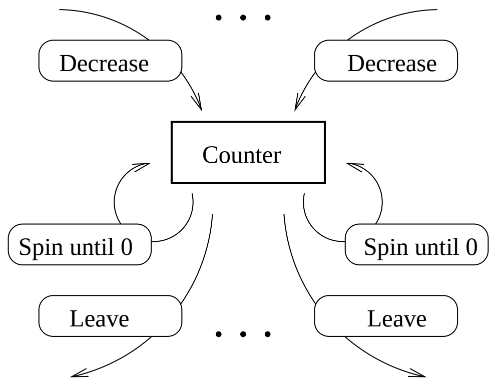

Fig. 3. Barrier implemented with fetch-and-add

```
        bne-    0b          \n\
" : "=&b"(result), "=&r"(tmp) : "r" (mem), "Ir\
"(val) : "cr0", "memory");
  return result;
}
```

The first instruction will create a reservation for use by the second instruction. The store can only be successful when the reserved bit is set. If not, the program will repeat the process. In this way, the procedure prevents two threads from updating the counter at the same time.

The advantage of this implementation is obvious. It has direct hardware support and is quite simple in terms of coding. It does not cost much memory either.

As we will find out, the performance of this design is acceptable only for a parallel system with a small number of processors, for example, less than eight nodes. We will see that as the number of processors increases, the memory contention increases sharply.

3.2 Distributed counter

Since the fetch-and-add design has a high contention cost as the number of processors increases, we would like to coordinate threads using multiple memory locations to reduce contention.

As a result, we produced a new implementation we call a *distributed counter* shown in Figure 4. Instead of allocating one counter in the shared memory, we allocate multiple counters as a byte array. The size of the array is equal to the number of threads in the barrier operation and the value of each element in the counter is initialized to one.

Similarly to the fetch-and-add barrier design, each thread arriving at the barrier point will decrease the counter. However, unlike the fetch-and-add design, they will decrease only their own element of the counter. In this way, there is no need for the fetch-and-add function because simultaneous decrementing of the counter cannot happen.

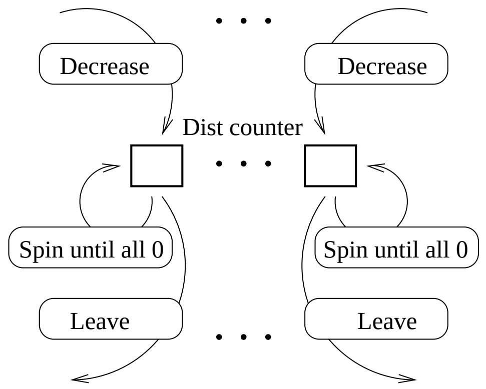

Fig. 4. Barrier implemented with distributed counter

In the spinning phase, each thread reads all the elements of the distributed counter array to make sure that all of the threads have decremented their own elements, thus arrived at the barrier point.

The idea of using different memory locations was also discussed in [12]. One difference here is that in their implementation the array element is increased by each thread continuously to mark the "milestones" of the synchronization.

From the test data shown later, we can see that this barrier design outperforms the fetch-and-add design and requires only a modest increase in memory (proportional to number of threads). With a 32-processor POWER4 system, we did not incur severe memory cost. As in the fetch-and-add design, we can assign multiple counters for multiple barriers, simplify the implementation.

3.3 Distributed counter with padding

One problem with the distributed counter design as described in the previous section is that false sharing can happen between elements of the counter array which reside on the same cache line. The result is that contention happens in the memory system even though counter storage is not strictly shared. False sharing can be eliminated by allocating the counter such that no two counter elements reside on the same cache line as shown in Figure 5.

For each thread, the operation sequence is exactly the same as the distributed counter design. The only thing different in the design is the distributed counter itself. The data structure is padded corresponding to the size of a cache line (128 bytes in a POWER4 system).

In section 4, we will see that this further reduces the time needed by a barrier operation. The drawback of this scheme is that the memory impact will be amplified by the unused padding space. Among the 128 byte cache line, only one byte is used.

We solve this problem by setting up two counter arrays in each parallel region and allow all of the barriers in the same parallel region to share these two counter sets.

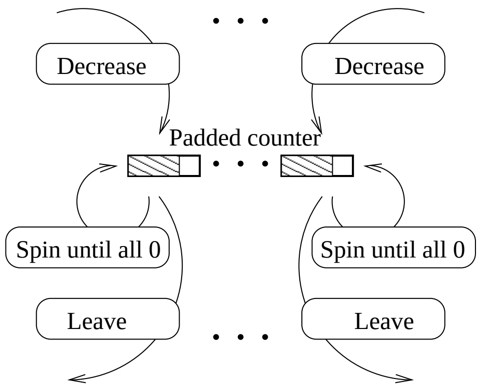

Fig. 5. Barrier implemented with padded distributed counter

This will reduce the memory consumption, while taking full advantage of a padded distributed counter.

To reuse the same data structure, we need to reinitialize the counter elements back to one after a synchronization. In case the program encounters multiple barriers in a small period of time, like the EPCC test. We need make sure that when we reinitialize the counter for the second barrier, we do not contaminate the previous one.

Suppose we have a very fast and a very slow thread, when the fast thread is free and encounters another barrier right away, it needs to reinitialize the counter before it can decrease the counter as designed. If this is the same counter as the one used in the previous barrier, the slow thread may still spin on checking whether all of the counter elements are zero. If the bit is reinitialized to one, the slow thread will not leave the first barrier nicely.

By having two counters, the threads can always initialize the alternative counter while leaving the current one, knowing that no threads are spinning on checking that one. Otherwise the current counter elements can not be all zero.

We can further reduce the memory cost by merging the two arrays into one position by using different bytes available in a cache line. If two barriers are not too close to each other, this won't affect the overall performance, but will halve the memory cost.

3.4 Combined with local sensor

Another interesting design was exploited in [7] and [8]. In comparison to the fetch-and-add barrier, the difference is that the authors did not let the threads spin on checking the hot accessed counter, but instead used a separate local sensor.

A bit is allocated in the shared memory local to each thread, and initialized to one before a barrier starts. Each thread, after decreasing the counter with a fetch-and-add operation, spins on checking the local bit. The last thread, the one that decreases the counter to zero, will set all the sensor bits to zero, signaling the spinning threads that they are free to leave. Thus, the counter is read many fewer times in the latter part of the barrier synchronization.

We implemented this barrier in our test environment and the performance data is available in the next section.

Finally, we can combine this idea with what we have in previous designs to create a new barrier scheme.

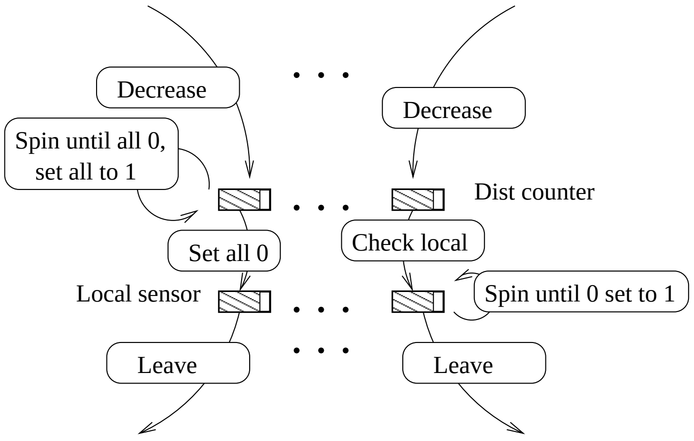

Fig. 6. Combined barrier with distributed counter and local sensor

In Figure 6, we have both a padded distributed counter and a local sensor. The local sensor is the same as the distributed counter, implemented as an array of cache lines.

Before a barrier starts, all the elements of the counter array will be set to one, as will the local sensor counter array. We let one thread in the group, for instance the master thread, act as if it is the last thread. It will decrease its own element of the distributed counter array and then spin to check whether all of the counter elements are zero. The rest of the threads will decrease their own counter elements and then spin on checking their own local sensors.

When the designated thread finds the counter elements are all zero, it will set all the counter elements back to one and then zero all of the elements in the local sensor array. Finally, when all of the threads leave the barrier after their local sensor is zeroed, they reset their local sensor back to one.

Algorithm 1 describes this more formally[2].

We would like to avoid allocating two cache line arrays for every barrier. In our test code, we use the same idea to reduce memory cost as we did for padded distributed array by letting all the barriers in a parallel region share the same pair of counter and sensor.

[2] Note that we did not emphasize the data structure here, they can be either a padded one or not, or even share the same cache line as we discussed previously.

Algorithm 1: Barrier with distributed counter and local sensor

```
Data : Distributed counter with each element as one
Data : Local sensor with each element as one
begin
    Decrease my own distributed counter element;
    if I am the master thread then
        repeat
            foreach element in distributed counter do check if it is zero
        until all distributed counter elements are zero;
        foreach element in distributed counter do
            set it back to one
        end
        foreach element in local sensor do
            set it to zero
        end
    else
        repeat
            check my local sensor element
        until it is zero;
    end
    Set my own local sensor element back to one;
end
```

Unlike the previous situation, we do not need two groups of counter here. The reason is that the last thread resets the counter values before any thread leaves the barrier, and each thread will reset its own sensor right after it is free.

In the combined barrier case, even if the fast thread is already spinning on checking its sensor for the second barrier, its counter element value will not affect the slow thread. This is so because, by the time the fast thread can decrease its counter element, the slowest thread must have passed the phase of resetting all the counter elements. The counter reset operation is done by the last thread (the slowest one) before it frees the remaining threads from the first barrier. In the worst case, the slowest thread will still be spinning on its sensor for the first barrier when the fast thread is spinning on its sensor for the second barrier.

In order to save memory, we could also merge the counter and the sensor into the same cache line using a different byte position. However, this would increase the barrier overhead (largely nullifying the benefit of the local sensors) as the counter and the sensor will be accessed at the same time in the same synchronization.

4 Performance data and analysis

We measured the EPCC benchmark using various barrier designs to show the differences in performance overhead.

Figure 7 shows the barrier overhead for each kind of implementations on a 32-way POWER4 system with varying numbers of threads. We used different labels to tag different designs. *FetchAndAdd* is for the first and the simplest barrier design. *DistCounter* is used for the barrier with distributed counter. *DistCounterPad* is used for the barrier with padded distributed counter. *LocalSensor* is for the barrier using local sensor. And the last one, *Combined* represents our new barrier scheme, the barrier with distributed counter combined with local sensor.

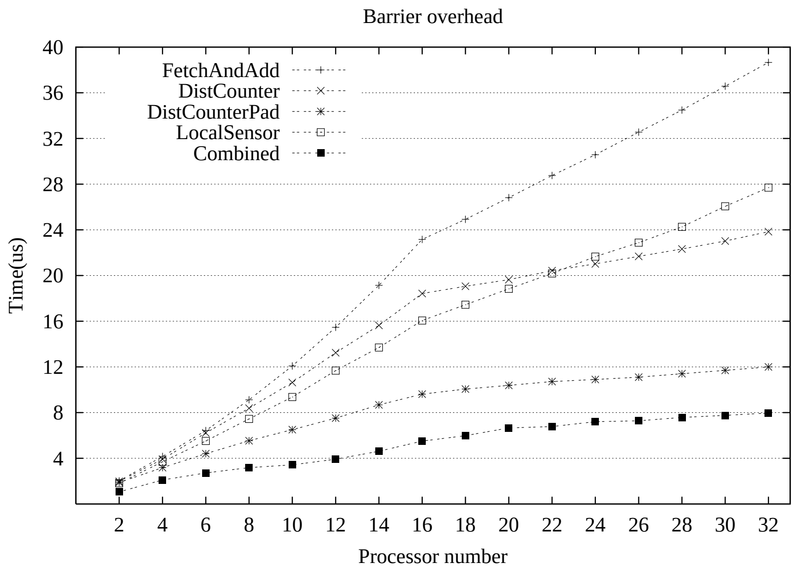

Fig. 7. POWER4 barrier overhead

To further understand the behavior of these barrier designs, we repeated the tests on a 16-way 375MHz POWER3 system. Figure 8 shows the overheads on POWER3.

Although POWER4 and POWER3 have different memory architectures[13], one can see that there is little difference in the relative overhead for different barrier designs. Of course, we could not compare the scalability of the designs beyond 16 threads on the POWER3 system so a completely equitable comparison was not possible.

From the figures we can see that the scalability of the overhead is not exactly the same, even considering the difference of the clock speed of the processors — recall that the POWER4 system we used is 1.1GHz and the POWER3 is 375MHz. The memory system in POWER4 system certainly gives it extra advantage.

To be able to compare the different implementations, we examine the performance counters on the more interesting POWER4 system, using `pmcount` program available on AIX 5.1. The program will print out the values of the different performance counters on each processor, when monitoring a given execution program.

First we do a simple measurement for the fetch-and-add barrier, using the exact same setting as we measure overhead previously. We check the performance counter for

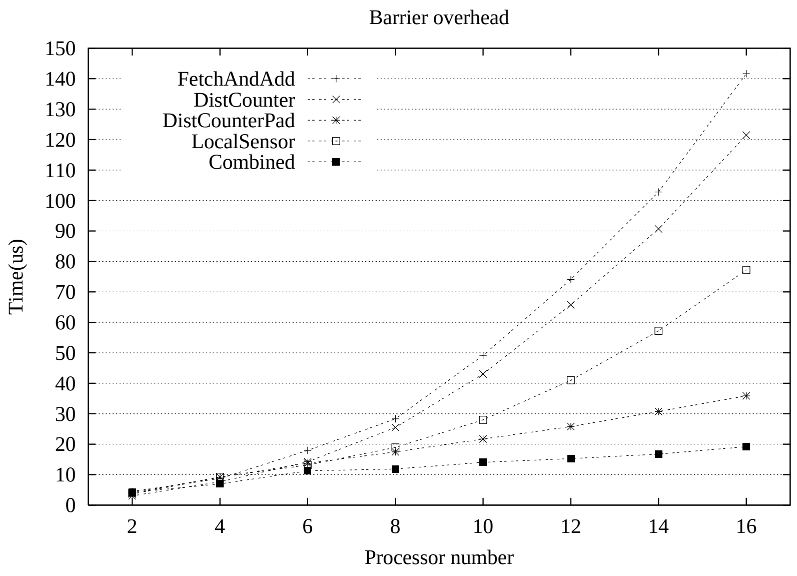

Fig. 8. POWER3 barrier overhead

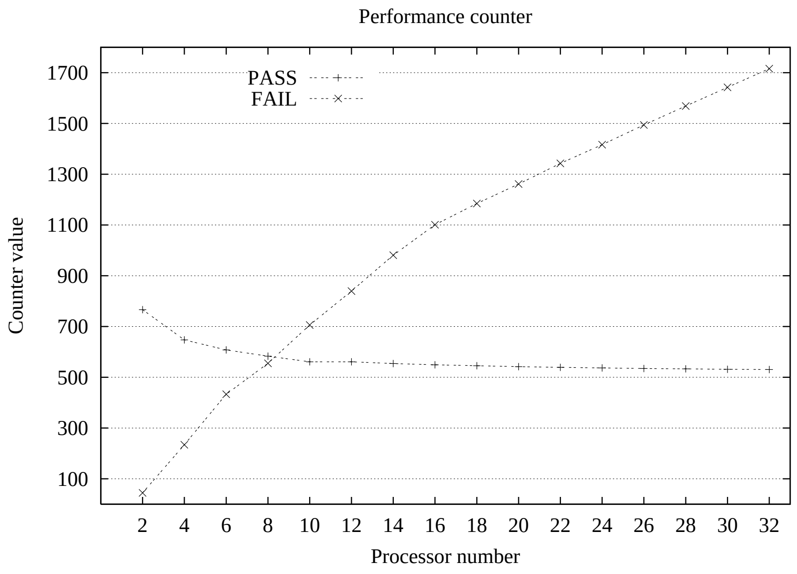

Fig. 9. Store conditional instruction pass and fail

store conditional instruction to see the relationship between contention and the number of processors involved in a barrier operation. Figure 9 shows the average number of pass and fail for the store conditional instructions on each processor.

As one can see, while the number of pass remained almost the same(in theory, it should be identical since we have the same number of barrier for different number of processors, we consider it is measurement error here), the number of fail increased sharply as more processors are added. That explains why we can not have a scalable performance for fetch-and-add barrier.

Similarly, we can get the performance counter for cache misses, representing data traffic. An interesting case here is L2 miss for distributed counter.

For a POWER4 system, an L2 miss may be served by L2 from another L2 in the same MCM, L2 in a different MCM, L3 in the same MCM and L3 in a different MCM. We put them all together as L2 misses. Figure 10 pictures average L2 misses on different processors for barrier with distributed counter and padded distributed counter.

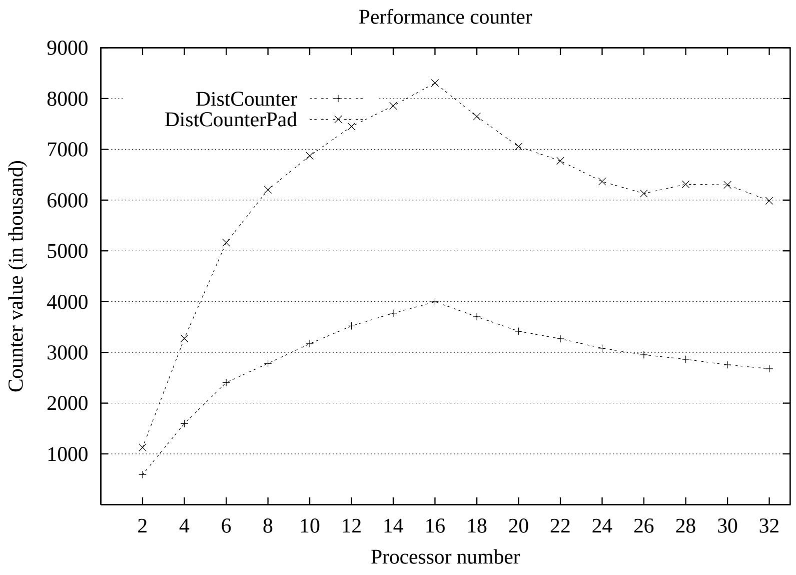

Fig. 10. L2 misses

To our surprise, there are more cache misses in the barrier with padded counter, even though it cost less time. We are not sure what exactly caused this, but we suspect that the cache coherent algorithm used in hardware cause extra contention when different processors accessing the same cache line, which out weighs the traffic overhead caused by L2 misses.

5 Summary and future work

Barrier synchronization is a well-studied topic, and an important one in parallel programming. In this paper, we studied the performance characteristics of several alternative implementations of barrier synchronization on modern, complex shared memory multiprocessors.

We have implemented several different barrier schemes and measured their performance on the shared memory POWER3 system and the distributed shared memory POWER4 system. We analyzed barrier performance data through timing and hardware performance counters. In the future, we wish to study the impact of different barrier implementations on other shared memory platforms particularly looking at issues of non-uniform memory access and scalability.

Through the introduction of the *distributed counters with local sensor* implementation, we have demonstrated a dramatic 79% reduction in overhead on a 32-way POWER4 system when comparing to a fetch-and-add implementation and a 33% improvement on the same system over the next best technique, distributed counters with padding. While these experiments were done using the explicit barrier test in EPCC microbenchmarks, we get similar improvements in the performance of the implied barriers terminating work-sharing constructs and parallel regions. The availability of a low overhead barrier also provides more freedom to the compiler to automatically introduce parallelism in serial code.

6 Acknowledgments

Roch Archambault (from the IBM Toronto Lab) had useful discussions with us; Raul Silvera and Shimin Cui (from the IBM Toronto Lab) gave us valuable technical assistance during the integration of the new barrier scheme to the existing SMP runtime frame work.

7 Trademarks and copyright

References

1. Message Passing Interface Forum. MPI: A message-passing interface standard, 1994.
2. V.S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–339, December 1990.
3. Arvind Krishnamurthy and Katherine A. Yelick. Optimizing parallel programs with explicit synchronization. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–204, 1995.

4. OpenMP Architecture Review Board. OpenMP specification FORTRAN version 2.0, 2000. http://www.openmp.org.
5. OpenMP Architecture Review Board. OpenMP specification C/C++ version 2.0, 2002. http://www.openmp.org.
6. Edinburgh Parallel Computing Center. OpenMP microbenchmarks, 1999. http://www.epcc.ed.ac.uk/research/openmpbench.
7. John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. on Computer Systems*, 9(1):21–65, February 1991.
8. Dimitrios S. Nikolopoulos and Theodore S. Papatheodorou. A quantitative architectural evaluation of synchronization algorithms and disciplines on ccNUMA systems: The case of the SGI Origin2000. June 1999.
9. Steve Behling et al. The POWER4 processor introduction and tuning guide. Technical Report SG24-7041-00, International Technical Support Organization, November 2001. ISBN 0738423556.
10. J. M. Bull. Measuring synchronization and scheduling overheads in OpenMP. In *First European Workshop on OpenMP*, October 1999.
11. IBM Technical Disclosure Bulletin. Barrier Synchronization Using Fetch-and-Add and Broadcast. 34(8):33–34, 1992.
12. Rainer Kreuzburg. Method of synchronization, 2001. United States Patent, No. US 6,330,619.
13. Stefan Andersson et al. RS/6000 scientific and technical computing: POWER3 introduction and tuning guide. Technical Report SG24-5155-00, International Technical Support Organization, October 1998.