Optimizing Primary Data Caches for Parallel Scientific Applications: The Pool Buffer Approach[1]

Liuxi Yang and **Josep Torrellas**

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

Abstract

Optimizing on-chip primary data caches for parallel scientific applications is challenging because different applications exhibit different behavior. Indeed, while some applications exhibit good spatial locality, others have accesses with long strides that prevent the effective use of cache lines. Finally, other applications cannot exploit long lines because they exhibit false sharing. To help processors execute these three types of applications efficiently, we introduce the *Pool Buffer*, a small direct-mapped cache accessed in parallel with the primary cache. The function of the pool buffer is to fetch long sectors of relatively short cache lines from memory on a miss, while only letting into the cache the lines that the processor actually references. The pool buffer can also perform sequential prefetching of sectors.

An evaluation of the pool buffer based on simulations of five 32-processor Perfect Club codes yields encouraging results. Adding a pool buffer of one-quarter the size of the cache causes a small increase in area while usually achieving large reductions in execution time. For example, for a range of caches with 32-byte lines, the execution time decreases by an average of about 20%. We also show that small 1-Kbyte buffers are often large enough to get most of the potential benefits. Finally, caches with pool buffers are more effective than caches with long lines and no pool buffer.

1 Introduction

High-performing memory hierarchies are essential to achieve good processor utilization in scalable shared-memory multiprocessors. Indeed, these machines are equipped with fast microprocessors that place considerable stress on the memory system with high requests rates. One way to handle these high request rates is for the processor chip to include primary caches that service most of the requests fast. These caches, however, have to be designed carefully. Indeed, because primary caches have to be very fast, they are necessarily small. To prevent their small size from inducing high miss rates, their design needs to be optimized.

Optimizing on-chip data caches for parallel scientific applications is particularly challenging because applications exhibit widely different behavior. On the one hand, some applications have good spatial locality and, therefore, benefit from long cache lines. Other applications, on the other hand, access the data with long strides and, therefore, make poor use of long cache lines. Furthermore, these accesses can easily wipe out the contents of a small on-chip cache. Finally, other applications cause false sharing in the caches. Clearly, a plain primary cache does not handle most of these cases optimally.

Several hardware schemes have been proposed to address some of these problems. One approach is to combine relatively short cache lines with prefetching. In the simplest prefetching scheme, called sequential prefetching [3], a cache access or a cache miss triggers the prefetch of the next line or next several lines. With this approach, spatial locality can be exploited while minimizing false sharing. One problem with this approach, however, is that useless data is sometimes brought into the cache and, therefore, useful data may be displaced. There are other, more advanced forms of hardware prefetching that are more efficient. However, while they may bring fewer useless lines into the cache, they are more complex to implement and use more chip area.

A related scheme that has been proposed for uniprocessors is stream buffers [5, 9]. Stream buffers are usually placed off-chip. They identify vector strides and then prefetch vector elements that are expected to be referenced. However, they may not always recognize the stride of the vector access correctly and, for best results, they need to be organized as FIFO buffers. While they are very useful in uniprocessors, it is unknown how well they work for large multiprocessors, where vectors are likely to be shorter and where some misses are caused by data sharing. Furthermore, they may not work very well with non-vector data.

Finally, there are many other optimization schemes for primary data caches that are either specifically designed to eliminate a certain type of misses or that try to overcome a certain obstacle. For example, miss caches [5], victim caches [5], and selective victim caches [12] all try to eliminate conflict misses. Fetch caches [2] are useful for caches with multicycle access times, while two-level on-chip caches [6] try to increase the effective size of the on-chip cache without excessively increasing its average access time. Overall, while all these schemes are useful, they do not specifically focus on the problem that we mentioned above.

In this paper, we introduce a hardware-only solution to the problem indicated above. We propose the *Pool Buffer*, a small direct-mapped cache that is accessed in parallel with the primary data cache. The function of the pool buffer is to fetch long sectors of relatively short cache lines from memory on a miss, while only letting into the cache the lines that the processor actually references. The results show that adding a pool buffer of one-quarter the size of the cache causes a small increase in area while usually achieving large execution time reductions. For example, for a range of caches with 32-byte lines, the execution time decreases by an average of about 20%.

This paper is organized in three main sections: Section 2 describes the concept of pool buffer, Section 3 discusses the experimental system that we use to evaluate the pool buffer, and Section 4 performs the evaluation.

[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; and NASA Contract No. NAG-1-613.

2 The Pool Buffer

The pool buffer is a small on-chip direct-mapped cache that is accessed in parallel with the on-chip primary data cache. The idea behind the pool buffer is simple: to increase the amount of contiguous data that a memory access brings on-chip while, at the same time, prevent the pollution of the primary cache with any data that is not currently useful. Such data is kept in the pool buffer for future use. In addition, the pool buffer organization keeps false sharing low. In the following, we present how the pool buffer works, the rationale behind the pool buffer, and a brief qualitative comparison to other schemes.

2.1 How the Pool Buffer Works

To enable the exploitation of more spatial locality, each entry in the pool buffer is a sector, namely, a set of contiguous cache lines. A sector has an address tag and, for each of its lines, a presence bit. If this bit is set, the corresponding line is present in the pool buffer.

When the processor references a line, both the cache and the pool buffer are searched. In our protocol, there is no duplication of data between the primary cache and the pool buffer. Therefore, either the cache hits and the pool buffer misses, the opposite occurs, or both cache and pool buffer miss. If the cache misses and the pool buffer hits, the requested line is copied from the pool buffer to the cache and the corresponding presence bit in the pool buffer is reset. If, instead, both pool buffer and cache miss, the pool buffer issues a request for the line that, in the best case, brings the whole sector. Once the data arrives, the requested line is stored in the primary cache. The remaining valid lines in the message are left in the pool buffer and marked as present only if they are not already in the primary cache. Note that this may imply overwriting parts of the sector already present in the pool buffer. We allow this to keep the design simple.

If the line request issued by the pool buffer reaches the directory, the directory assembles the whole sector in state *shared* and sends it to the pool buffer. However, if the request hits in the secondary cache, the secondary cache may supply an incomplete sector. Two cases are possible. The first case occurs in systems where the secondary caches are organized in sectors like the pool buffer. In such case, the secondary cache supplies the sector, which contains the requested line plus maybe some other lines. The sector is transferred such that all of its valid lines are in state *shared*. If necessary, the secondary cache may have to perform a write back to memory in the background.

The second case occurs in systems where the secondary caches are not organized in sectors. In this case, the secondary cache simply supplies the line to the primary cache. The data is transferred in whatever state it is in. Overall, we perform these two incomplete sector transfers to avoid having to wait for a directory access.

Note that we are insisting that the pool buffer keep data in state *shared* only. Since the lines in the pool buffer are visible to the cache coherence protocol, this is done to keep the design simple. For example, consider a write miss on a sector present in other caches. In this case, the request reaches the directory. We do not want the directory to invalidate the whole sector from the other caches and bring it to the requester in state exclusive. This would be similar to having a cache line as long as the sector and would create false sharing. Furthermore, we do not want to bring the line missed on in state exclusive and the rest of the lines in the sector in state shared. This would complicate the protocol. Instead, what we want is to bring to the requester the whole sector in state shared. This implies that the directory assembles the sector by requesting a write back from all processors that have parts of this sector in state dirty. Once the sector is assembled, it is sent to the requesting processor. The requesting processor then retries the original write. This time, however, the write will issue invalidations to copies of the line (not the whole sector) in other caches. Overall, while this combined transaction is inefficient in several ways, we do it this way for simplicity reasons. Furthermore, this transaction does not occur frequently. Indeed, write misses are not as common as read misses followed by a write. The latter type of transaction pair is cheaper.

From this discussion, we see that a line can only leave the pool buffer in either of three cases. The first one is when the processor accesses the line. In this case, the line is moved to the primary cache. The second case is when the line is invalidated by another processor writing on it. In this case, the rest of the lines in the sector are not affected. This approach minimizes false sharing. The third case occurs when the line is displaced from the buffer due to a pool buffer conflict. In this case, all the lines in the sector are displaced.

Figure 1 shows a simple block diagram of the primary data cache, pool buffer, and CPU. The arrows indicate the direction of data transfers. Since the pool buffer is smaller than the cache, it has a smaller access latency and, therefore, detects a hit or a miss before the cache does.

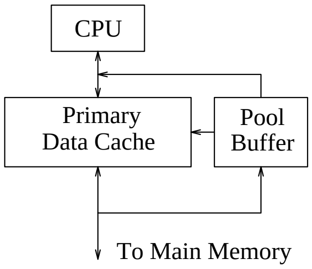

Figure 1: Block diagram of the primary data cache, pool buffer, and CPU.

To further improve the performance of applications with spatial locality, we optionally add sequential prefetching to the pool buffer. This is done by adding a two-bit counter to each sector in the pool buffer. When the sector is first loaded into the pool buffer, the counter is cleared. Then, every time that a line in this sector is moved from the pool buffer to the cache, the counter is incremented. If the counter reaches a certain threshold, the pool buffer issues a prefetch for the next sector. The complete algorithm used by the primary cache and pool buffer is as follows. The variable *prefetch* is true if prefetching is enabled.

```
Case hit in cache:
    Break;
Case miss in cache and hit in pool buffer:
    Copy the line from the pool buffer to the cache;
    Reset the line's presence bit in the pool buffer;
    If (prefetch)
      Increment the counter in the referenced sector;
      If (counter == threshold)
        Prefetch the next sector;
        Set the new sector's counter to 0;
        /* only valid lines in the sector that are not
           already in the primary cache can be loaded
           into the pool buffer with presence bit = 1 */
    Break;
Case miss in both cache and pool buffer:
    Pool buffer issues a request for the line;
    If (secondary cache hits)
      Bring the (possibly incomplete) sector;
    Else
      Bring the sector;
    Place the requested line in the cache;
    Place the remaining lines, if any, in the pool buffer;
    /* only valid lines in the sector that are not
```

Table 1: Characteristics of the applications.

Application	What It Does	Lines (Thous.)	Refs. (Mill.)
TRFD	Solves 2-electron integrals	0.3	15.0
QCD	Solves lattice gauge problems	2.6	24.2
DYFESM	Analysis of symmetric anisotropic structures	8.2	32.3
ARC2D	Solves a 2-D fluid dynamics problem	5.5	153.3
FLO52	Analysis of 2-D transonic airflow	4.0	29.3

```
    already in the primary cache can be loaded
    into the pool buffer with presence bit = 1 */
If (prefetch)
  Set the sector's counter to 0;
Break;
```

2.2 Rationale

With the pool buffer, we address the three types of behavior exhibited by scientific applications that we indicated above:

- For applications with spatial locality, the data is efficiently brought to the first level of the memory hierarchy by the long sectors in the pool buffer. Applications with spatial locality are further helped with the addition of prefetching.
- Long sectors and prefetching do not necessarily harm applications with long stride if relatively short cache lines are used. Indeed, while the pool buffer may receive many useless cache lines, the primary cache loads only the lines that it uses. Consequently, primary cache pollution is minimized.
- Similarly, false sharing is minimized if relatively short cache lines are used because each cache line in a sector can be separately invalidated.

2.3 Comparison to Other Schemes

To finish this section, we compare the pool buffer to two other related schemes that improve the performance of the primary cache.

- *Sequential Prefetching* [3]. The main advantage of a non-prefetching pool buffer over access- or miss-triggered sequential prefetching of cache lines is that the primary cache is not polluted with cache lines that will not be used. In addition, if the request issued by the pool buffer reaches the directory or in systems where the secondary caches are organized in sectors, a second advantage is lower network bandwidth requirements. Indeed, loading a sector in one message is less disruptive that loading several, smaller-sized lines as in sequential prefetching.
- *Stream Buffer* [5]. One limitation of the stream buffer is that, for the most part, it is targeted to vector accesses only. Furthermore, it may not always recognize the stride of the vector access. Another limitation is that, for best results, stream buffers are organized as FIFO buffers, which makes them more expensive. Overall, while stream buffers work well for uniprocessors, it is unknown how well they work for large multiprocessors, where vectors are likely to be shorter and where some misses are caused by data sharing. In addition, the stream buffers have to be visible to the cache coherence protocol. It is true, however, that a stream buffer is easier to include in a machine because it is usually placed off-chip and, therefore, does not require redesigning the chip.

3 Experimental Setup

To evaluate the performance impact of adding pool buffers, we have used detailed execution-driven simulations. In this section, we describe the applications used, the simulator, and the evaluation methodology.

3.1 Applications

We trace the parallel versions of the five Perfect Club fortran codes [1] presented in Table 1. These versions run with 32 processors and were parallelized using a parallelizing compiler and later by hand [4]. Loop iterations are statically assigned to processors in chunks. Since the codes take a long time to run, we reduced their time requirements while preserving their parallelism and reference behavior. We did this by reducing the number of iterations rather than the data set sizes where possible. Table 1 shows the number of lines of code and the number of references simulated in each application. We trace array data references only.

3.2 Simulation System

For our simulations, we use the EPG-sim execution-driven simulator [11]. We start by compiling our applications with the Parafrase-2 compiler [10]. For each array access in the program, a pass in the compiler introduces a call to the simulator. Each call takes as arguments the address of the data accessed, the processor ID, the type of access, and the time-stamp. For the time-stamp, it is assumed that each basic operation like an arithmetic operation takes one cycle to execute. The applications are then linked with the EPG-sim memory system simulator. Finally, the applications are executed. At run time, the applications call the simulator, which simulates the memory system.

The architecture modeled by the simulator is a 32-processor cache-coherent UMA shared-memory multiprocessor (Figure 2). The machine uses a plain directory-based invalidate protocol. Each processor has one private cache (Section 4.1 and 4.2) or a primary and a secondary private cache (Section 4.3). In addition, each processor has an optional pool buffer and is connected to memory via an omega network. Both caches and pool buffers are direct-mapped. Memory accesses take different number of cycles depending on the cycle time of the processor. The latter is in turn determined by the access time of the primary caches. For example, for 32-Kbyte caches with 32-byte lines, a contention-free read miss takes 72 cycles for data clean in memory and 144 cycles for data dirty in another processor. Since we use the release consistency model, write misses do not stall the processors. Read misses, however, stall the processors. In our simulations, we accurately model the contention in the system. For the network, we use the analytical delay model for indirect multistage networks presented in [7]. The channels of the network are 64-bit wide.

As indicated before, both primary cache and pool buffer are accessed at the same time. If the access hits in any of the two, the data is returned to the processor in one cycle. In our design, the bandwidth between the pool buffer and the

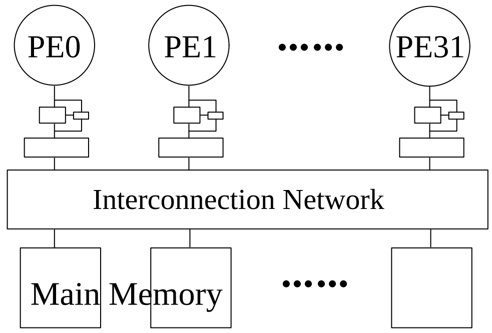

Figure 2: Architecture modeled in the simulations.

primary cache is such that a cache line can be transferred from the former to the latter in one cycle. Finally, when a sector is loaded into the pool buffer, the data is brought into the buffer in 8-byte chunks per cycle. As each valid line is brought in, the primary cache checks for a match in its tags. If there is a match, the data is marked as not present in the pool buffer.

3.3 Evaluation Methodology

To evaluate a primary cache configuration, we use two metrics, namely the area of the cache configuration and the execution time of the application using that cache. To determine the area, we use the model proposed by Mulder *et al* [8]. The model gives the area in register-bit equivalents (or *rbe*s). Figure 3 shows the *rbe* cost for different cache configurations. The figure shows two sets of two curves: the set at the bottom corresponds to plain caches, while the set at the top corresponds to caches with pool buffers. For each set of two curves, the topmost curve corresponds to 8-byte cache lines while the bottom curve corresponds to 32-byte lines. In all cases, caches and pool buffers are direct-mapped and have the same line size. Pool buffers are one fourth as large as caches, and pool buffer sectors contain four lines. Finally, the curves corresponding to caches plus pool buffers plus prefetching are practically the same as those for caches plus pool buffers. For this reason we have not shown them.

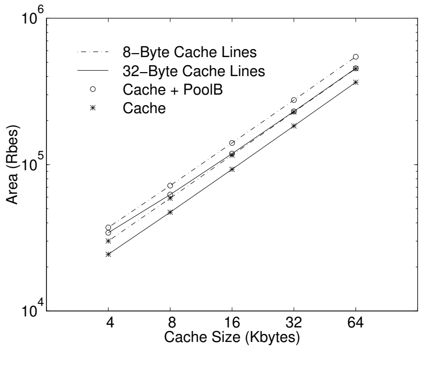

Figure 3: Area used by different direct-mapped cache configurations. From bottom to top, the sets of two curves correspond to plain caches and caches with pool buffers respectively.

From the figure, we see that adding the pool buffer to the primary cache does not increase the area much. Except in 4-Kbyte caches, the new area is about one fifth or less larger than the old one. The reasons for the small change are the small size of the buffer and the relatively few bits used for tags in the long sectors of the buffer. Prefetching does not add significantly more cost because of the simplicity of the prefetching scheme.

To assess the execution time, we start by using a model proposed by Wada *et al* [13] that gives the cache access time in *ns* as a function of the cache size, line size, associativity, and other RAM organization parameters. However, since Wada's model is based on the somewhat old 2μ CMOS technology, we scale the results down so that we can get the primary cache access time for 0.5μ CMOS technology. Then, we multiply the resulting number by 1.2 to account for gate precharging times to obtain the CPU cycle time. The cache access times and CPU cycle times that we obtain for different sizes of direct-mapped caches are shown in Figure 4. Again, we show curves for 8-word and 32-word cache lines.

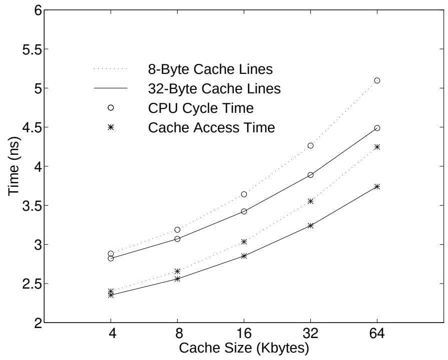

Figure 4: Cache access time (bottom two curves) and CPU cycle time (top two curves) for different configurations of direct-mapped caches.

To compute the cache access time, we do not consider any effect of the pool buffer. This is because the pool buffer is faster than the cache and is accessed in parallel with it. The higher speed of the pool buffer is due to its smaller size and the fact that, because of its long sectors, it has relatively less tag space than the main cache. In fact, the access time of a pool buffer of about one quarter the size of the cache is less than about 75% of the cache access time. Therefore, in our on-chip memory system (Figure 1), the pool buffer responds with either a hit signal or a miss signal and a precharge of the cache data output path long before the cache access finishes. As a result, in our simulations, caches with and without pool buffers have the same cycle time.

In our simulations, when we vary the cache configuration, both the cache access time and the CPU cycle time change. However, since the main memory and network architecture do not change, the time in *ns* required to service a read miss from memory remains constant. This time is set to a number of *ns* equivalent to 72 CPU cycles for 32-Kbyte caches with 32-byte lines. Since our simulations are based on cycles, before starting a simulation with a given cache configuration, we need to determine the cache hit and the cache read miss time in cycles. To determine the cache read miss time in cycles, we use the aforementioned read miss time in *ns* and the CPU cycle time from Figure 4. The cache hit time in cycles is always 1. With this information we can run the simulation. After the simulation, we multiply the number of cycles that the simulation took by the CPU cycle time from Figure 4 to obtain the overall execution time.

4 Evaluation

Using the methodology described, we have performed detailed simulations whose results we discuss in this section. In our discussion, we compare three environments, namely plain caches (*Cache*), caches with pool buffers (*C+Pool*), and caches with pool buffers and prefetching (*C+P+Pref*). Unless indicated otherwise, caches and pool buffers are direct-mapped and have the same line size; pool buffers are one fourth as large as caches and have sectors with four lines; finally, prefetching is performed with a threshold of two (Section 2). These are the base parameters.

In the following, we first compare the three environments using the base parameters. Then, we explore some parameter variations. For simplicity, all these experiments are performed using a single cache per processor. Finally, we examine a system with primary and secondary caches. In our experiments, we use caches with 32-byte lines. In addition, to see the effects of false sharing, we also use 8-byte lines.

4.1 Impact of the Pool Buffer and Prefetching Support

To determine the impact of the pool buffer and prefetching support, Figure 5 shows, for different cache configurations, the execution time per instruction of the application (TPI) as a function of the area required by the cache configuration. The TPI is found by dividing the application execution time by the number of instructions, a constant for each application. For each application, the leftmost chart compares the *Cache* and *C+Pool* environments, while the rightmost chart compares the *C+Pool* and *C+P+Pref* environments. Each chart has curves for 8- and 32-byte cache lines. Each curve has five data points which correspond, from left to right, to 4-, 8-, 16-, 32-, and 64-Kbyte caches respectively.

Focusing first on the leftmost charts, we note two main points. The first one results from comparing the *Cache* and *C+Pool* environments for the same cache and line sizes. The charts show that adding a pool buffer causes only a small increase in area while achieving consistent and often large execution time reductions. Indeed, TPI reductions occur in all cache configurations, and range from 8% to 59%. The reductions tend to be more prominent for small caches since these are the caches with most of the misses. Similarly, the reductions are larger for short cache lines. This is because longer cache lines exploit spatial locality a bit better while suffering little false sharing. Overall, therefore, adding a pool buffer appears attractive.

A second observation reinforces this conclusion. Indeed, let us again compare a *Cache* and a *C+Pool* curve for the same line size. However, in *Cache*, we examine a cache twice as large as in *C+Pool*. This means, for example, that we compare the third data point starting from the left in the *Cache* curve to the second data point in the *C+Pool* curve. We see that, in the large majority of cache configurations, adding a pool buffer is cheaper in area and more effective in reducing execution time than doubling the size of the cache. The reduced area requirements of the former environment are due to the smaller size of the pool buffer and its low tag cost. Overall, in terms of TPI reduction, adding a pool buffer is on average 22% better than doubling the cache size.

We now consider the impact of prefetching. From the rightmost charts, we see that the support for prefetching has little area cost. However, it is also clear that, for 3 out of 5 applications, it does not have any impact on the TPI. Prefetching has an impact only on the TPI of *ARC2D* and *FLO52*: it reduces it by an average of 17%. The reason for this is that *ARC2D* and *FLO52* have numerous conflict misses. The other applications have fewer conflict misses. In addition, the sharing misses of the latter applications have poor spatial locality. Given that the prefetching scheme works only if there is spatial locality in the misses, it is understandable that prefetching does not work for *TRFD*, *QCD*, and *DYFESM*. Overall, however, given the small cost of adding prefetching support, even if prefetching may not have a large impact, we still recommend supporting it.

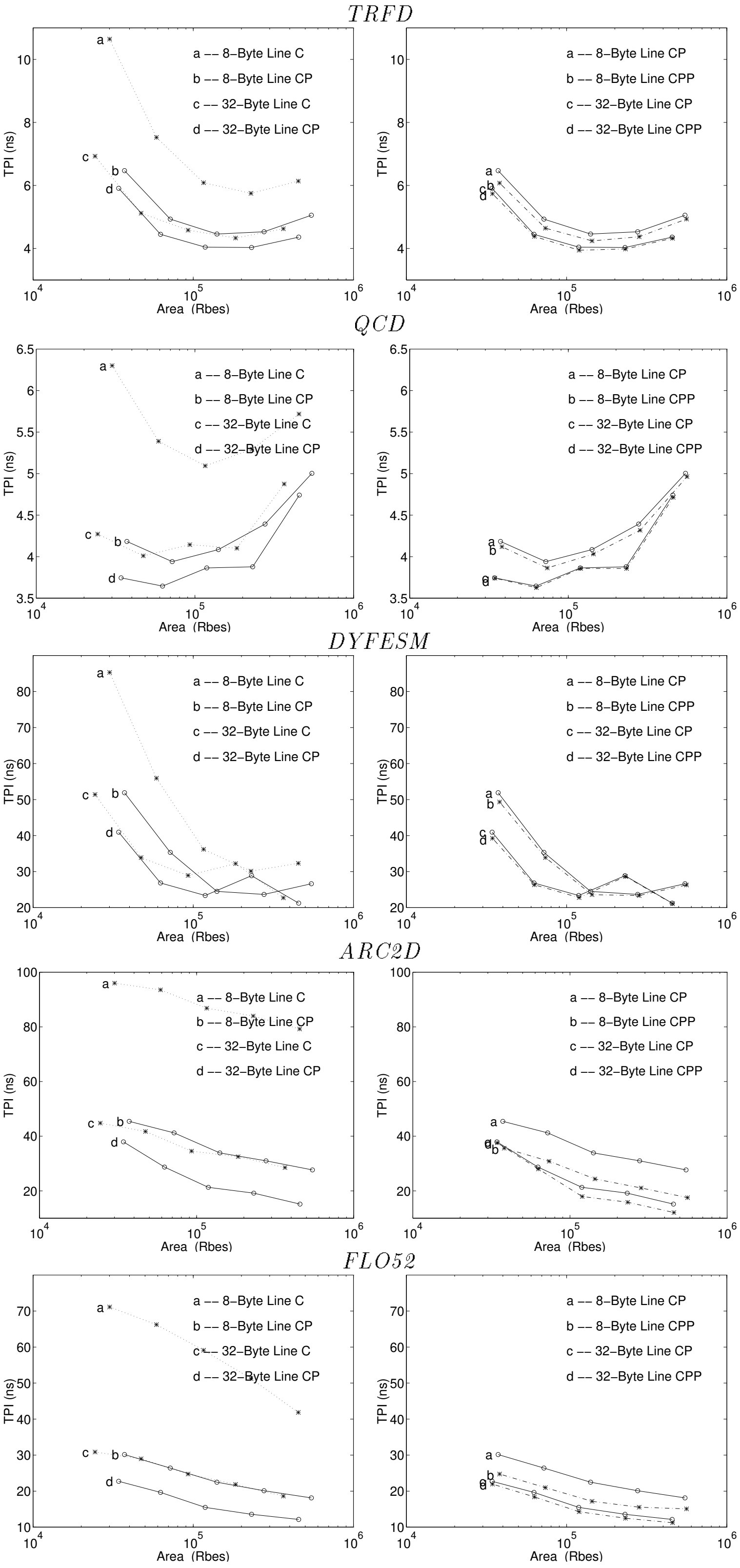

Figure 5: Time per instruction (TPI) as a function of the area requirements of the cache configuration. For each application, the leftmost chart compares the *Cache* and *C+Pool* environments, while the rightmost chart compares the *C+Pool* and *C+P+Pref* environments. Due to space limitations, the environments are labeled *C*, *CP*, and *CPP* for *Cache*, *C+Pool*, and *C+P+Pref* respectively.

4.1.1 Analysis of the Execution Time

To understand the variation of the TPI in Figure 5 better, we now decompose the execution time into its component categories. Figure 6 shows, for each application, the execution time under different environments with 8-byte cache lines (leftmost charts) and 32-byte lines (rightmost charts). In each chart, we consider five different cache and pool buffer combinations, given by the pairs in parenthesis. For example, (4,1) refers to a 4-Kbyte cache and a 1-Kbyte pool buffer. For each of these cases, we consider a *Cache*, a *C+Pool*, and a *C+P+Pref* environment, referred to as *C*, *CP*, and *CPP* respectively in the figure due to lack of space. Execution times are broken down into time spent executing instructions (*busy*), time waiting for another processor to finish executing a serial section (*idle*), time waiting in locks and barriers due to contention or load imbalance (*sync*), and read miss stall time (*read*). Recall that write misses do not stall the processor. For a given application, all bars are normalized to the leftmost bar in the 8-byte chart. Of all these categories, the pool buffer and prefetching can reduce *read* and, indirectly, *idle* and *sync*. Indeed, *idle* is reduced by decreasing the *read* time of the processor executing the serial section. Similarly, reducing *read* may decrease load imbalance and therefore reduce *sync*.

Looking first at the leftmost charts, we see that, for the majority of the applications and environments, the pool buffer eliminates most of the *read* time. In particular, for large caches in *TRFD* and *QCD*, little more can be done. If larger speedups are not achieved in these two cases, it is because the *busy* time increases with the cache size. For the other cases, a strategy that eliminated misses that do not have spatial locality would complement the pool buffer. However, little further gains should be expected. Overall, for 8-byte cache lines, pool buffers speed up the applications by an average of 40.2%.

The rightmost charts in the figure show why the pool buffer has a lower impact on cache configurations with 32-byte lines. Indeed, the pool buffer removes a smaller fraction of the *read* time and, in addition, the *read* time is smaller to start with. Therefore, the speedups are smaller. It is also clear from the charts that these applications do not have an inordinate amount of false sharing. A likely manifestation of false sharing, namely a *read+idle* time for 32-byte lines as large or larger than that for 8-byte lines, occurs only for *DYFESM* in the *(32,8)* case. Overall, for 32-byte cache lines, pool buffers speed up the applications by an average of 20.3%.

Finally, as indicated above, prefetching has a significant impact only in *ARC2D* and *FLO52*, where conflicts cause a number of sequential misses. Even in these applications, for 32-byte cache lines, the reduction in execution time is only 8.4%.

4.2 Exploring Parameter Variations

While the previous section has given us an idea of the cost and performance impact of the pool buffer, we have used only the base parameters. In this section, we consider two issues: we vary the size of the pool buffer for a fixed cache size and we compare caches with pool buffers to caches with long lines and no pool buffers.

The effect of varying the size of the pool buffer for a fixed cache size is shown in Figure 7. For each application, the figure shows the execution time under 5 different cache configurations with either 8-byte or 32-byte cache lines. The 5 cache configurations consist of 16 Kbyte caches with either no pool buffers, or with 1-, 2-, 4-, or 8-Kbyte pool buffers. No prefetching is performed. For a given application, all the bars are normalized to the leftmost one and are decomposed

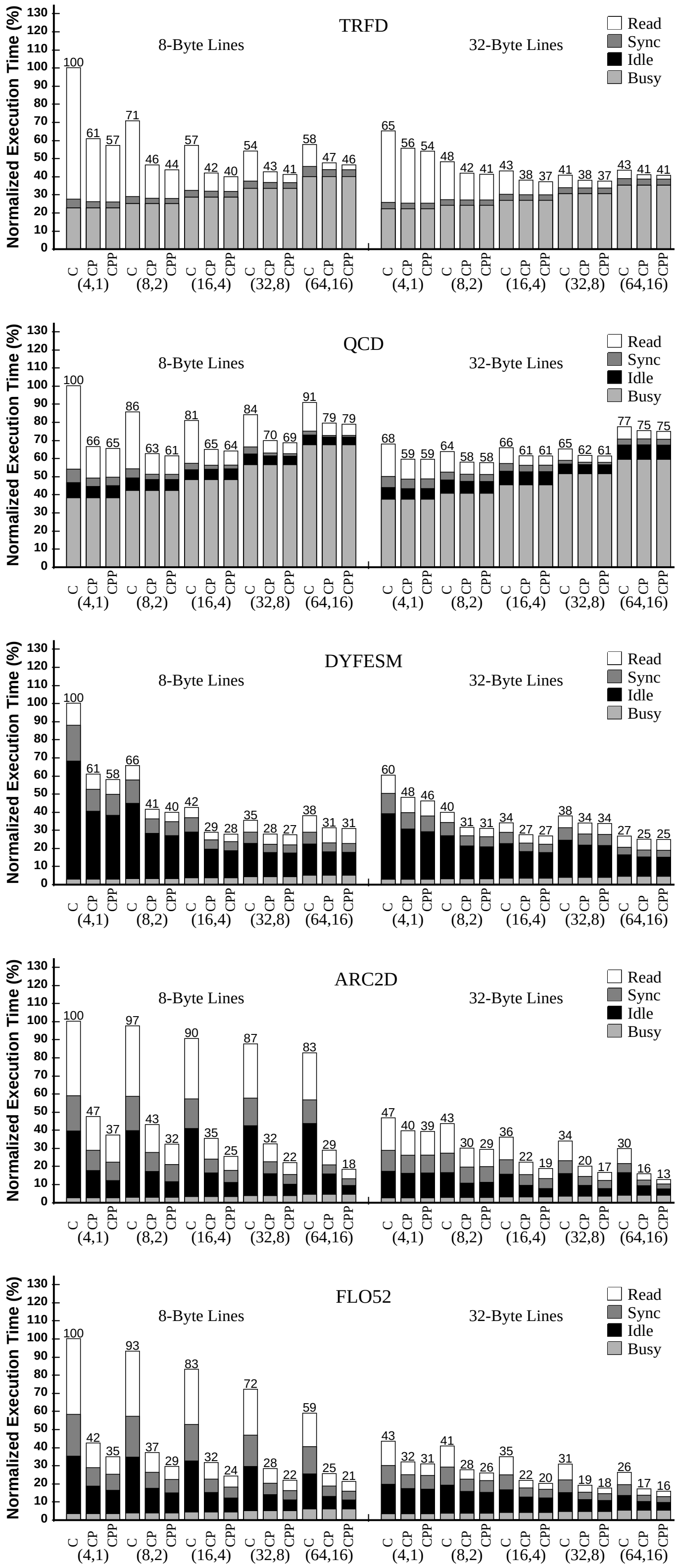

Figure 6: Decomposition of the execution time for the *Cache*, *C+Pool*, and *C+P+Pref* environments. Due to space limitations, the environments are labeled *C*, *CP*, and *CPP* respectively.

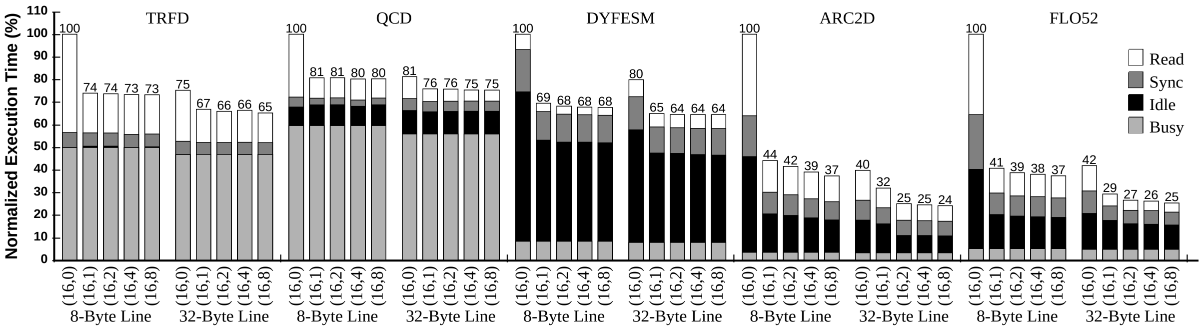

Figure 7: Varying the size of the pool buffer for a fixed cache size. The notation (16,4) refers to a 16-Kbyte cache plus a 4-Kbyte pool buffer.

into the categories discussed above.

The figure shows that the execution time drops significantly by adding even a small 1-Kbyte pool buffer. In addition, the execution time is not very sensitive to the actual size of the pool buffer for the range considered. This suggests that what is important is to capture a set of critical cache lines that are referenced and missed on with high spatial locality. These lines fit in a 1-Kbyte cache and their spatial locality is captured by the buffer sectors. It is possible that even smaller pool buffers would perform reasonably well. However, 1 Kbyte is already small enough for us not to study the issue further.

Still, it can be seen that the size of the pool buffer has some impact. In *TRFD*, *QCD*, and *DYFESM*, which have a smaller working set size, the size of the pool buffer matters little. However, in *ARC2D* and *FLO52*, which have a larger working set, larger buffers are better. Overall, therefore, we conclude that, at least for 16-Kbyte primary caches, a small buffer of 1 Kbyte is enough to get most of the potential benefits. However, if there is extra area available on-chip, it is better to add a larger pool buffer since it will produce better speedups for applications with a large working set.

The second issue that we examine is how caches with pool buffers compare to caches with long lines and no pool buffers. Figure 8 compares the execution time of two systems with 16-Kbyte caches. For each application, the leftmost bar corresponds to no pool buffer and 128-byte cache lines, while the rightmost bar corresponds to 4-Kbyte pool buffers, 32-byte cache lines, and 4 lines per sector. In both cases, the data is loaded on-chip in 128-byte chunks. For each application, the bars are normalized to the no pool buffer case.

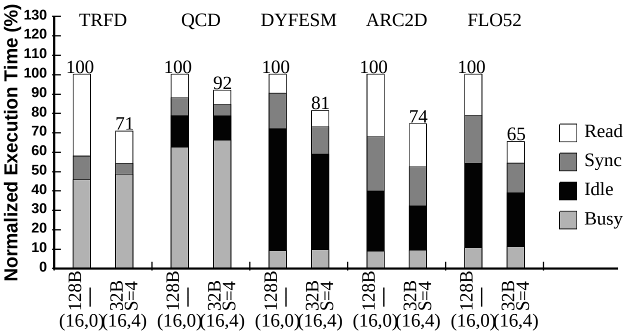

Figure 8: Comparing caches with long lines and no pool buffers to caches with pool buffers.

The figure shows that, in all the applications, 32-byte lines and pool buffer perform better than 128-byte lines and no pool buffer. The execution time is reduced by an average of 23.4%. While the system with the pool buffer has a slightly higher *busy* time due to a higher cache access time, it is able to eliminate more *read* time (and, indirectly, *idle* and *sync* time) and therefore runs faster. The system with the long lines suffers from a combination of false sharing and conflicts in the primary cache. Therefore, the pool buffer approach is preferable.

4.3 Systems with Secondary Caches

To see the impact of adding a pool buffer to more sophisticated memory hierarchies, we model a system with primary and secondary data caches per processor. The secondary cache has sectors of size equal to those in the pool buffer. The size of the lines and the number of lines per sector is also equal to those in the primary cache. The cache coherence protocol used is such that a secondary cache miss loads the whole sector into the secondary cache, while a write invalidates the written line only. The number of cycles required by the processor to read a word from the secondary cache is 12 cycles.

Figure 9 shows the execution time for several systems with 8-Kbyte and 64-Kbyte direct-mapped primary and secondary caches respectively. From left to right within one application, these systems have no pool buffer, a 2-Kbyte pool buffer, and a 2-Kbyte pool buffer plus prefetching. The bars are decomposed into the usual categories and, for each application, are normalized to the no pool buffer system. In the figure, if we compare the *Cache* and *C+Pool* systems, we see that adding the pool buffer reduces the execution time of the applications by an average of 19.8%. This figure is only slightly smaller than the one obtained for similar-sized primary caches and no secondary caches. Indeed, it can be seen from Figure 6 that the reduction in execution time for the latter configuration was 21.3% on average then. A slightly smaller impact now makes sense because secondary caches tend to reduce the contribution of memory access stall to the total execution time.

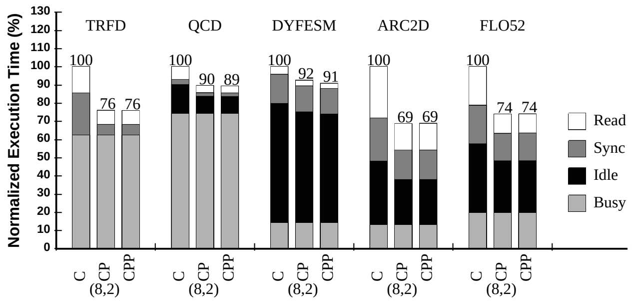

Figure 9: Effect of the pool buffer on the execution time for systems with secondary caches. Due to space limitations, the *Cache*, *C+Pool*, and *C+P+Pref* systems are labeled *C*, *CP*, and *CPP* respectively.

If we compare the *C+P+Pref* systems to the *C+Pool* systems we see that, like before, prefetching has only a very minor effect. Overall, therefore, we conclude that, qualitatively, the effect of pool buffers in systems with secondary caches is similar to that in systems without secondary caches. As expected, the actual speedup obtained is a bit smaller in the systems with secondary caches.

In addition to the simulations shown in this paper, we have performed other simulations varying pool buffer and cache parameters. Overall, for 32-byte cache lines, the best sector sizes tend to be about 4-8 lines in 4-Kbyte pool buffers and 4 lines in 1-Kbyte pool buffers.

5 Conclusions

Optimizing on-chip primary data caches for parallel scientific applications is challenging because of the different behavior that different applications exhibit, for example good spatial locality, long-strided accesses, or false sharing. As a solution to this problem, this paper has introduced the *Pool Buffer*, a small direct-mapped cache that is accessed in parallel with the primary cache. The pool buffer fetches long sectors of relatively short cache lines from memory while only letting into the primary cache the lines that the processor actually references.

An evaluation of the pool buffer based on detailed execution-driven simulations of five 32-processor Perfect Club codes produced encouraging results. Adding a pool buffer of one-quarter the size of the primary cache causes a small increase in area while usually achieving large execution time reductions. For a range of caches with 32-byte lines, the execution time decreases by an average of about 20%. We also showed that small buffers of 1 Kbyte are often enough to get most of the potential benefits. Finally, caches with pool buffers are more effective than caches with long lines and no pool buffer.

Overall, we expect that primary data cache configurations will become more important as processors get faster. Given that there is only a small amount of on-chip area available for the primary cache, it is important to utilize it well. Pool buffers are one way of effectively utilizing some of this area.

Acknowledgments

We thank the anonymous referees, David Koufaty, Zheng Zhang, and the graduate students in the I-ACOMA group for their feedback. Josep Torrellas is supported in part by an NSF Young Investigator Award.

References

[1] M. Berry et al. The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. *International Journal of Supercomputer Applications*, 3(3):5–40, Fall 1989.

[2] B. Bray and M. Flynn. Fetch Caches. Technical Report CSL-TR-93-561, Computer Systems Laboratory, Stanford University, February 1993.

[3] F. Dahlgren, M. Dubois, and P. Stenstrom. Fixed and Adaptive Sequential Prefetching in Shared-Memory Multiprocessors. In *Proceedings of the 1993 International Conference on Parallel Processing*, pages I:56–63, August 1993.

[4] R. Eigenmann, J. Hoeflinger, G. Jaxon, and D. Padua. The Cedar Fortran Project. Technical Report 1262, Center for Supercomputing Research and Development, October 1992.

[5] N. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[6] N. Jouppi and S. Wilton. Tradeoffs in Two Level On-Chip Caching. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 34–45, April 1994.

[7] C. Kruskal and M. Snir. The Performance of Multistage Interconnection Networks for Multiprocessors. In *IEEE Trans. on Computers*, pages 1091–98, December 1983.

[8] J. M. Mulder, N. T. Quach, and M. Flynn. An Area Model for On-Chip Memories and its Application. In *IEEE Journal of Solid-State Circuits*, pages 98–106, February 1991.

[9] S. Palacharla and R. Kessler. Evaluating Stream Buffers as a Secondary Cache Replacement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 24–33, April 1994.

[10] C. D. Polychronopoulos et al. Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors. In *Proceedings of the 1989 International Conference on Parallel Processing*, volume II, pages 39–48, August 1989.

[11] D. K. Poulsen and P.-C. Yew. Execution Driven Tools for Parallel Simulation of Parallel Architectures and Applications. In *Proceedings of Supercomputing '93*, pages 860–869, November 1993.

[12] D. Stiliadis and A. Varma. Selective Victim Caching: A Method to Improve the Performance of Direct-Mapped Caches. Technical Report UCSC-CRL-93-41, University of California, Santa Cruz, October 1994.

[13] T. Wada, S. Rajan, and S. Przybylski. An Analytical Access Time Model for On-Chip Cache Memories. In *IEEE Journal of Solid-State Circuits*, pages 1147–1156, August 1992.