Speeding up Irregular Applications in Shared-Memory Multiprocessors: Memory Binding and Group Prefetching[1]

Zheng Zhang and **Josep Torrellas**

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

Abstract

While many parallel applications exhibit good spatial locality, other important codes in areas like graph problem-solving or CAD do not. Often, these irregular codes contain small records accessed via pointers. Consequently, while the former applications benefit from long cache lines, the latter prefer short lines. One good solution is to combine short lines with prefetching. In this way, each application can exploit the amount of spatial locality that it has. However, prefetching, if provided, should also work for the irregular codes. This paper presents a new prefetching scheme that, while usable by regular applications, is specifically targeted to irregular ones: *Memory Binding* and *Group Prefetching*.

The idea is to hardware-bind and prefetch together groups of data that the programmer suggests are strongly related to each other. Examples are the different fields in a record or two records linked by a permanent pointer. This prefetching scheme, combined with short cache lines, results in a memory hierarchy design that can be exploited by both regular and irregular applications. Overall, it is better to use a system with short lines (16-32 bytes) and our prefetching than a system with long lines (128 bytes) with or without our prefetching. The former system runs 6 out of 7 Splash-class applications faster. In particular, some of the most irregular applications run 25-40% faster.

1 Introduction

Effective cache utilization is often key to high performance in scalable shared-memory multiprocessors. A major contributor to an application's effective use of the cache is good spatial locality of reference. Indeed, if the application has good spatial locality, long cache lines can be used to intercept many references and thereby, keep the miss rate low.

Unfortunately, important engineering applications in areas like graph problem-solving, event-driven simulation, VLSI design, or tree-handling often exhibit poor spatial locality. Often, the dominant data structures or objects in these applications are small records no more than a few tens of bytes in size. Furthermore, these objects are frequently interconnected and accessed via pointers. For example, Splash's *Pthor* application [16] is an event-driven simulator whose data structures are small records representing gates or nets. Gates and nets are linked by pointers to form the circuit. In this and similar applications that we refer to as irregular, a processor often accesses two or three fields in a record and then dereferences a pointer. Consequently, references are widely scattered.

One consequence of this behavior is that long cache lines like those over a hundred bytes long proposed for scalable multiprocessors, do not work well for all applications. Irregular applications run more efficiently with short cache lines. Indeed, short lines bring less useless data into the cache and, therefore, cause less traffic. In addition, they are likely to cause fewer cache conflicts and less false sharing [17]. It seems, therefore, that if we want to design general purpose computers, we need to accommodate these two contradictory demands on the cache line size.

Several techniques have been proposed to address this problem. One of them is to have an adjustable cache line size [7]. The idea is to let the hardware adapt to changing object sizes by automatically merging or splitting cache lines dynamically. A second approach is to combine short cache lines and adaptive prefetching [6]. In adaptive prefetching, every miss fetches a certain number of contiguous lines. The hardware then monitors the use of the lines. If they are used, more lines will be fetched in future misses; if they are not used, fewer lines will be fetched per miss. Again, this scheme adapts to the object size. Unfortunately, both schemes fail to use any object information that could be extracted, often very easily, from the source code. Instead of using help from the compiler or programmer, all is left to the hardware.

Two other techniques, regions [3] and block transfers [19], use the opposite approach, namely they use the help of the software to identify objects. However, given the overheads involved, these techniques are designed for large objects. In addition, identifying the regions or blocks and annotating the application is likely to require significant programmer involvement. Finally, these schemes do not exploit the fact that objects are often linked together with pointers.

Overall, the most intuitive solution to this problem is to use short lines combined with data prefetching. In this case, each application can exploit the amount of spatial locality that it has.

If prefetching support is provided, however, we would like it to work for both regular and irregular applications. Unfortunately, it is hard for prefetching to work well for our irregular applications. On the one hand, traditional prefetching hardware is effective for fixed-stride access patterns only [1, 4]. Therefore, it will not work very well for irregular, pointer-based codes. On the other hand, purely software-based prefetching driven by the compiler [14] does not seem applicable either: the compiler analysis required is very complex. Finally, software-based prefetching with hand-inserted prefetches is a possibility. However, for irregular applications, it is hard and very time consuming to get good speedups with hand-inserted prefetches [13].

This paper addresses the prefetching problem by proposing a new prefetching scheme that, while usable by applications with good spatial locality, is specifically targeted to irregular applications. We call the scheme *Memory Binding* and *Group Prefetching*. The idea is to bind in hardware and prefetch together groups of data that the compiler or programmer suggest are strongly related to each other. Examples are the different fields in a record or two records linked by a perma-

[1]This work was supported in part by the National Science Foundation under grants NSF Young Investigator Award MIP 94-57436 and MIP 93-08098; by NASA Contract No. NAG-1-613; and by a grant from Intel Corporation. Their support is gratefully acknowledged.

nent pointer. The support for this scheme is shared among the hardware, compiler, and programmer. This scheme, combined with short cache lines, addresses the purpose of the paper: a memory hierarchy design that can be effectively used by both applications with good spatial locality and irregular applications. Overall, it is better to use a system with short cache lines (16-32 bytes) and our prefetching than a system with long cache lines (128 bytes) with or without our prefetching. The former system runs 6 out of 7 Splash-class 32-processor applications faster. In particular, some of the most irregular applications run 25-40% faster.

This paper is organized as follows: Section 2 presents an overview of the scheme; Section 3 discusses the applications and their groups; Section 4 presents the necessary architectural support; Section 5 evaluates the scheme; and Section 6 compares it to other prefetching schemes.

2 Overview of Our Scheme

As indicated in the introduction, many important engineering applications often have two characteristics: first, their dominant data structures are small records no more than several tens of bytes in size; second, these records are frequently interconnected and accessed via pointers. In these applications, which we refer to as irregular, the programmer has provided a strong suggestion as to which data logically go together: the data in a record. In many cases, this is true and the different fields of a record are indeed accessed close in time. Therefore, we would like to somehow "bind together" in hardware the fields of the record into a group and then prefetch the whole group when one field of the record is accessed. This scheme we call *Memory Binding and Group Prefetching*.

Obviously, the compiler can identify these groups from record declarations in the source code. In addition, the compiler may be able to determine which fields of the record are never or seldom referenced and keep them out of the group. Similarly, if the programmer knows the code, he can point out to the compiler the fields of the record that are rarely accessed and therefore not worth keeping as part of the group. If either of these two scenarios is true, the compiler can then try to rearrange the fields within a record so that the fields that do belong to the group are bunched together. In all cases, at the end, the compiler always asks the programmer for confirmation before binding the group. At that point, the programmer can enable or deny the binding.

For non record-based, more regular codes, the compiler may be able to find groups even if they are not declared as records. A good example are the columns of the main matrix in Splash's *Ocean* application, which have obvious access patterns. In this case, the compiler will identify that the column is accessed as a group and simply ask the programmer for confirmation before binding the group. Finally, a third possibility is for the programmer to directly instrument the code, explicitly defining a group. This is useful in non record-based programs when the compiler alone cannot figure out the groups. We will see an example in Splash's *Cholesky* application.

While records are hints of inter-related data, permanent pointers between records are strong hints of groups that go together. Consider, for instance, *Maxflow* [18], a graph problem-solving application. The nodes and edges in the graph are represented by records. The record of a node and the records of the edges connected to it are linked by permanent pointers in the application. In the code, when a processor accesses a node record, it is likely that it will soon follow one of these permanent pointers to access the record of an edge connected to the node. Therefore, the compiler can identify each of these permanent pointers and ask the programmer for confirmation to make it a *prefetching* link. While prefetching links usually are permanent pointers, the programmer may determine that even a pointer that is changed several times in the code actually links highly-connected groups together. In this case, the programmer can direct the compiler to instrument the code at every point where the pointer is changed to make it a prefetching link. In any case, when a processor accesses a group, the hardware will automatically prefetch all the groups it points to via prefetching links. Note that we only prefetch groups *directly* connected to the group being accessed. That is, we follow only one level of prefetching links. This strategy we call *Pipelined Prefetching of Groups*.

The architectural support for these optimizations is easy to integrate with modern designs of scalable shared-memory machines that include a network processor per node. As we will see in Section 4, when a memory module is accessed, its network processor will perform the operations necessary to forward whole groups to the requesting processor. A feature of our scheme is that, by spreading the costs into relatively simple hardware changes, simple compiler support, and programmer feedback, it reduces the overall burden on the system. Furthermore, if there is no compiler or programmer input and therefore no prefetching occurs, the extra hardware that we have added does not slow down the machine.

Finally, compared to software prefetching with hand-inserted prefetches, our scheme has four advantages: it does not have the instruction overheads of prefetches or address generation for prefetches, it saves bandwidth by not issuing so many prefetch messages, it does not require the programmer to understand the code as much, and it issues prefetches earlier when traversing a list of records linked with pointers. We perform a more complete comparison of our scheme to existing prefetching schemes in Section 6.

3 Applications and Their Groups

To gain more insight into the idea of groups, this section examines the applications that we use in our evaluation and the groups that they contain. The applications include most of the Splash [16] applications plus one pointer-intensive application called *Maxflow* [18]. The fraction of pointer-chasing references varies quite a bit among the applications. Using this metric, we classify *Barnes*, *Maxflow*, *Pthor* and, to a lesser extent, *MP3D* as pointer-intensive or irregular applications. They often access structures of records via pointers. The rest of the applications are more array-based and, therefore, have more regular access patterns. All the applications are parallel and run with 32 processors except for *Maxflow* which, due to its small size, runs with only 16 processors. All applications are run to completion.

For each application, Table 1 shows the size of the code, the number of shared data references issued by the application, what the code does, the input data used, the most important groups and their size, and the type of group. Groups can be of three types: *Comp*, *Feed*, and *User*. *Comp* groups are those that a hypothetical compiler could easily identify and determine whether they are beneficially grouped. Usually, they are easy to identify because they are declared as records. The compiler decides whether the records are beneficially grouped based on the program access patterns (i.e. whether the fields in the group are clearly accessed together).

Feed groups can also be identified by the compiler. However, the compiler is unsure whether it is beneficial to keep the data in groups. Either not all the fields in a record are accessed at the same time or it cannot be determined because they are accessed via pointers. This often occurs in pointer-based programs. Therefore, the compiler definitely needs programmer feedback on whether or not to create a group. In this case, the programmer must have some knowledge of the application.

Finally, *User* groups are those which the compiler cannot identify alone, usually because they are not declared as records or arrays. Examples of these groups are subsections of

Table 1: Characteristics of the applications.

Applic.	Lines of Code (Thou.)	Shared Data Refs. (Mill.)	What It Does	Input Data	Main Groups and Size in Bytes	Type of Group
Water	1.4	112.3	Simulates evolution of set of water molecules	343 molecules for 4 steps	Water molecule (700)	*Comp*
Ocean	1.5	110.2	Simulates eddy currents in an ocean basin	Grid of size 98	Column of the working matrices (784)	*Comp*
Cholesky	2.3	21.4	Cholesky factorization of a sparse matrix	bcsstk14.0 matrix (1806 columns)	Column (500 on average)	*User*
MP3D	2.2	3.9	Simulates a rarefied hypersonic flow	10,000 particles for 10 time steps	Particle (36); cell (40)	*Comp/User*
Barnes	3.6	56.8	Simulates the evolution of galaxies	1000 bodies	Body (100); cell in the tree (70)	*Feed*
Maxflow	1.2	36.7	Determines the maximum flow in a directed graph	20-ary 2-cube graph	Graph node (40); graph edge (40)	*Feed*
Pthor	11.1	15.5	Simulates a digital circuit at the logic level	Small risc processor	Element (168); ptr to input and output nets in elem. (56); event list for each net (28); event (12); others	*Feed/User*

arrays whose elements are accessed together. The programmer has to write directives on the source code for the compiler to build the group. In some cases, as we will see, the programmer can find these groups very easily.

In the first application, *Water*, groups are records representing water molecules. The accesses to the different fields of a molecule record are clear and close enough in time to allow us to classify the groups as *Comp*. In *Ocean*, each processor works on several columns of several matrices at a time. Instead of grouping all these columns in one big group, we assign a different group to each column since we want to keep groups relatively small to minimize cache conflicts in small caches. The groups and access patterns are obvious enough for the compiler to recognize them. Therefore, we classify the groups as *Comp*. *Ocean* is written in Fortran and, therefore, the size of the columns is statically determined. Depending on the problem size solved, however, only a fraction of the allocated data will be used. Since the problem size is entered as an input parameter, our prefetcher may prefetch more data than necessary. The only way to eliminate this problem is for the programmer to tell the compiler the problem size that he will use in the runs.

In *Cholesky*, the groups are either individual columns of the matrix or sets of several columns called supernodes. We use columns because they are smaller. Since this is a sparse matrix and zeros are not stored, columns have a variable size. All columns are allocated in sequence in a large vector. Therefore, the compiler is not able to recognize each individual column. However, it is fairly easy, even for someone unfamiliar with the code, to recognize that these array sections are accessed as a group and instrument the code. Consequently, we classify the groups as *User*. In *MP3D*, the basic operation is a particle move. In the simplest case, this involves accessing three groups: the particle record, the cell where the particle was, and the cell to which the particle moves. The accesses are clear and close enough in time to classify the groups as *Comp*. In Section 4.3.1, we perform a transformation that requires some knowledge of the application. For this reason, we also classify the groups as *User*.

Finally, *Barnes*, *Maxflow*, and *Pthor* are all pointer-based applications. The data structures are declared as records and are therefore easy for the compiler to identify as groups. However, given the complex pointer access patterns, the compiler needs programmer feedback. Hence, the groups are *Feed*. In the case of *Pthor*, after analyzing the code, we found that several small groups can be further grouped into a larger one because they are accessed together. Therefore, we classify *Pthor*'s groups as *User* too. We note that it is relatively easier for the programmer to give feedback for *Barnes* and *Maxflow*: both are relatively simple codes. It is harder, however, to do it for *Pthor*: the code and its data structures are more involved. For the pipelined prefetching of groups, the prefetching links used are discussed in Section 4.3.1.

4 Architectural Support

To prefetch the groups identified by the compiler or programmer in the applications, we need to enhance our shared-memory multiprocessor with some special architectural support. In this section, we describe the architectural support required. We describe the support for three prefetching optimizations. The three optimizations complement each other and are presented in the order of decreasing cost-effectiveness. The description assumes a directory-based scalable shared-memory architecture with distributed memory modules. Each node has a network processor. In the following, we discuss each optimization in turn.

4.1 Memory Binding and Group Prefetching

Once the groups in the application have been identified, we must pass the information to the hardware. This is done as follows. Groups are marked in the source code with the keyword *Group*, which takes, as arguments, the beginning and end addresses of the group. Figure 1-(a) shows an example of a group composed of the six fields of a record. In addition, the main memory of the machine is augmented with two bits per memory line: the *next* (N) and *boundary* (B) bits. The *Group* keyword is translated into a system call that sets these two bits for the memory lines where the group is allocated. Alternatively, one could design an instruction that directly sets these bits without operating system involvement. In any case, for a given memory line, N is set to one if the next memory line contains data belonging to the same group as the data in this line. B is set to one if this line contains at least one boundary of a group. In Figure 1-(b), we show the N and B bits for the three memory lines where the *foo* group is allocated and the two surrounding lines. These bits are actually stored in the directory so that they are accessed faster. Obviously, they are set only when a group spans more than one memory line.

Once the *Group* system call is executed and the N and B bits set, prefetching is enabled. When a memory module receives a read request for a memory line, the network processor in that module checks if the N bit in the line is set. If N is set, the network processor performs memory reads to increasing addresses on behalf of the requesting processor until it reaches a memory line with $B = 1$. After that, if the line that was originally requested had (N = 1, B = 0) or (N = 0, B = 1), the network processor repeats the same procedure on decreasing addresses until B = 1 again. All these lines are sent in a pipelined fashion to the requesting processor. As an example, in Figure 1-(b), an ordinary read to line 1 triggers the prefetching of lines 2 and 3, while a read to line 2 triggers, in that order, the prefetching of lines 3 and 1. Finally, a read to line 3 triggers the prefetching of lines 2 and 1.

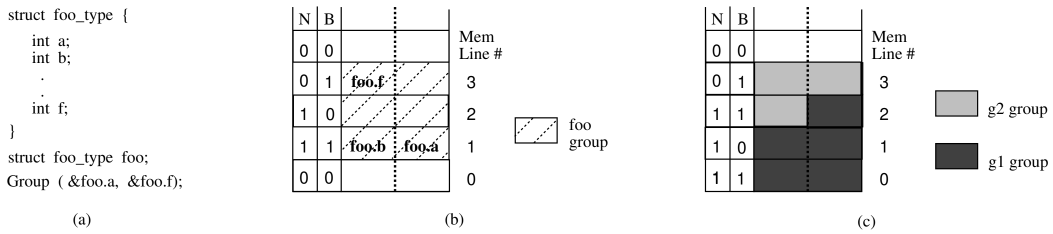


Figure 1: Binding several memory lines. The six fields in record *foo* are grouped together in Chart (a). The *Group* system call sets the N and B bits of the memory lines where *foo* is allocated as shown in Chart (b). Chart (c) shows two groups sharing a memory line.

The setting of the N and B bits we call *Memory Binding*, and this type of prefetching *Group Prefetching*. In our current design, the prefetching is triggered by read misses only, since read misses have more spatial locality than write misses. Similarly, in our current design, the directory sends all the prefetched lines in shared mode. Clearly, the directory may have to send write back requests to other processors before being able to send memory lines in state shared to the prefetching processor. When the network processor reads the N and B bits of a memory line from the directory, it also checks if the requesting processor is a sharer of the line. If so, the memory read is skipped. This reduces the resulting traffic. Finally, note that that these are simple operations that can be effectively pipelined: in the simplest case, for each prefetched memory line, the network processor performs a directory read and update and a memory read.

Page and memory line fragmentation are handled by the *Group* system call. If the group crosses a virtual page boundary, the N and B bits are set as if the parts of the group in the two pages were two completely different groups. A more common case is for different groups to share the same memory line. To handle this case, the scheme would need to keep an additional $logW$ bits in each memory line, where W is the number of words per line. If these $logW$ bits were not all zero, they would encode which of the $logW - 1$ word boundaries in the line was the boundary between the two groups; if the line contained more than one group boundary, we would randomly choose one boundary. Then, if the address read was lower than the boundary, we would prefetch decreasing addresses only; otherwise we would prefetch increasing addresses only.

In reality, however, we do not add the $logW$ bits. We can use the N and B bits and still prefetch correctly most of the time. We will assume that, when a processor misses on a memory line that is shared by two groups, it is accessing the beginning of the group allocated in higher addresses rather than the end of the group allocated in lower addresses. The reason is that, often, the programmer lays out the fields of a record roughly in the order in which they are accessed. Otherwise, as we discuss in Section 4.2, the layout of the fields may be changed to suit us. With this assumption, Figure 1-(c) shows that the N and B bits are sufficient. The figure shows two groups that share a memory line. For that memory line, we set N and B to one. With this encoding, a miss to lines 0 or 1 will bring, as desired, lines 0, 1, and 2, while a miss to lines 2 or 3 will bring, as desired, lines 2 and 3.

Since a processor may be working on several groups at the same time, group prefetching may force much data into its cache. To avoid displacements in small primary caches, our scheme sends the data to the secondary cache only.

4.1.1 Undoing the Binding

Sometimes, processors may neglect to access a subset of the fields in a group, or a group may exhibit false sharing. This may occur either because the access patterns in the program change with time or simply because the initial binding was wrong. Intuitively, therefore, it may seem desirable to be able to undo some of the memory bindings dynamically.

In our experiments, we have examined removing from a group a seldom-accessed or inconsistently-behaved part of it. To do so, we considered adding a *prefetch* bit (*pf*) to each line in the secondary cache. This bit is set when the line is group-prefetched into the cache and is reset when the processor accesses the line. With this support, a line that performs poorly is detected when the processor tries to access it and the corresponding cache entry has the correct tag, *pf* is still one, but the line is invalid. This means that the cache line has been invalidated between the prefetch and any access. When this situation was detected, we broke up the group into two groups by removing from the original group the bad memory line and the lines beyond it. This was done by having the processor, in the same miss request, ask the network processor to break up the group. The network processor simply changes the N and B bits for the bad line and its lower-addressed neighbor line. We have evaluated this scheme in our simulations. Fortunately, we have found that undoing the binding in this way improves the performance only slightly. For this reason, we do not add the *pf* bit and do not undo any binding.

4.2 L1 Pipelined Prefetching

While prefetching into the secondary cache can hide a large fraction of the miss penalty without inducing cache thrashing, processors still waste time missing in the primary cache. For this reason, we enhance the primary cache with a simple form of sequential prefetching that we call *L1 Pipelined Prefetching*. This prefetching gradually transfers the group from the secondary to the primary cache as it is being used. In this scheme, each cache line in the secondary cache stores the N bit discussed in Section 4.1. When a cache line is moved into the primary cache, its N bit is also brought in. When the processor accesses line L in the primary cache, if the line has $N = 1$, the cache will start prefetching line $L + 1$ from the secondary cache (if the line is not already in the primary cache) and reset N. Obviously, other similar schemes are possible. For example, more latency could be hidden by prefetching several lines at a time. Similarly, we could also have prefetched lines with decreasing addresses. The scheme proposed, however, is one of the cheapest.

This scheme has two minor shortcomings. Firstly, for best results, the different fields in a given group should be laid out in memory in the order in which they are accessed. Otherwise, the pipeline may break. We expect future compilers to sometimes change the order of the fields in data structure declarations to optimize the layout in this way. Alternatively, the compiler may be able to reorder statements in the code to optimize the order of accesses. These changes may not be possible in all programs. For example, the fields of a record may sometimes be accessed by adding constants to pointers, or the control flow may be too unpredictable. However, we feel that these changes are quite easily done for some programs. Indeed, frequently, the different fields of a record are accessed in only a few areas of the code and in a clear, consistent manner. In our work, we changed by hand the layout of groups in *Water* and exchanged some of the lines of code in *MP3D* and *Barnes*. We found this task very easy and moderately effective.

The second problem is that this prefetching scheme may bring data past the end of the group being operated upon. This occurs when the two groups share a memory line. For example, in Figure 1-(c), an access to group *g1* line 2 will also bring line 3 into the primary cache. To avoid this problem, we could store the B bit in the caches and use it in the prefetching logic. For simplicity, we do not do that.

4.2.1 Other, Not Cost-Effective Alternatives

In our analysis, we have also examined two alternatives to L1 Pipelined Prefetching. One scheme is to have the group prefetches described in Section 4.1 bring the data all the way to the primary cache. This scheme we call *First Prefetch* because, hopefully, the data in a group will be group-prefetched into the primary cache only once. Indeed, if primary cache conflicts cause misses on other parts of the group, the data will likely be supplied by the secondary cache; no other group prefetch from memory will be necessary. As expected, we had to reject this scheme because it causes many conflict misses in the primary cache, specially when the processor works on several groups at a time.

In the second scheme, called *Always Prefetch*, any miss in the primary cache triggers a group prefetch. This group prefetch comes either from the memory (if the request misses in the secondary cache), or from the secondary cache (if the request hits). The rationale for this is to refill the primary cache. Clearly, however, this scheme has many problems and, as a result, was also discarded. Firstly, the hardware support is very expensive because the secondary cache needs to know about groups and be able to supply the data in a group to the primary cache. Secondly, the performance is low because there are many conflicts in the primary cache as well as heavy traffic between the caches. Much of this traffic, in fact, contains data that has already been used up and is useless.

4.3 Pipelined Prefetching of Groups

Base group prefetching (Section 4.1) prefetches a whole group once an access to the group reaches memory. In this section, we expand the prefetch to include the groups that the group causing the miss points to with prefetching links. To avoid cache thrashing, we follow only one level of links: only the groups directly connected to the group causing the miss are prefetched. This scheme we call *Pipelined Prefetching of Groups*.

For a group to prefetch another, the pointer that points from the source group to the destination group has to be declared as a prefetching link. The declaration is inserted in the source code after the pointer is set. Figure 2-(a) shows an example. In the example, the *PrefetchingLink* system call ties $A{\rightarrow}next$ to group B so that a prefetch of A also prefetches B. The tie remains in place until another PrefetchingLink call forces $A{\rightarrow}next$ to be tied to another group instead.

Prefetching across groups is tricky because pointers must be dereferenced and, therefore, virtual addresses must be translated into physical ones. There are two ways of handling such translation. The first one is to let the prefetching engine have access to the TLB; the second one is not to. The first approach is easier because, when the prefetching engine wants to dereference a pointer, it only needs to examine the TLB and determine the physical address to use. This approach, however, has two problems. Firstly, since the TLB is inside the processor chip in most microprocessors, we need to modify the processor chip. Secondly, the prefetching engine must be very close to the CPU. Unfortunately, this likely means that the applicability is limited. Indeed, imagine that we are following the prefetching links shown in Figure 2-(b). Further, suppose that the prefetching engine is able to determine whether or not a word contains a prefetching link and, if so, issue the prefetch. If the prefetching engine is close to the CPU, in most cases it can only examine the words at the same time as the processor is accessing them. Therefore, the prefetches are likely to be issued not much earlier than when the data is actually needed. For example, in Figure 2-(b), when the processor accesses pointer P1, the prefetcher reads its contents and prefetches the second record. Unfortunately, the processor itself is likely to dereference P1 very soon and, therefore, not much latency will be hidden. This problem occurs when prefetching irregular pointer structures; regular access patterns can be easily learned and the data structures prefetched in advance [1].

Instead, to use off-the-shelf processors and to generate the prefetches earlier, we place the prefetcher in the memory system. More specifically, the prefetching logic is integrated in the network processors and in the mechanism that performs L1 Pipelined Prefetching between the secondary and primary caches. The scheme works as follows. When a memory module receives a read request for a line belonging to a group, the network processor group-prefetches the group, follows its prefetching links, and group-prefetches the groups they lead to. If these groups are in other memory modules, the network processor simply sends a read request to the appropriate memory module on behalf of the original processor. Similarly, when the L1 Pipelined Prefetching mechanism transfers a cache line between the secondary and primary caches, if the line contains a prefetching link, the prefetcher sends a group-prefetch request to the network processor. Note that, with these two architectural extensions, the prefetch for the second record in Figure 2-(b) is issued quite before the processor can get a hold of pointer P1.

For this scheme to work, we need two more architectural supports. First, we need to mark and detect prefetching links. This can be done by adding one bit per line in the memory and in the caches. This bit, called the pointer bit (P), is set if the line has at least one prefetching link. The second architectural support is for each network processor to have a table called Group Translation Table with the virtual-to-physical translation of the prefetching links that it owns. This is necessary because the network processors do not have access to the TLB. By convention, a network processor owns a prefetching link if the home memory of the group that is the source of the link is in the same node as the network processor. The group translation table is shown in Figure 2-(c). In the figure, word 2 of memory line 1 is a prefetching link to an group at virtual address 32. Therefore, the P bit in line 1 is set. In addition, the group translation table in the network processor has an entry with the line and word containing the link, the virtual address of the group pointed to by the link (32), and its physical address (16).

We can now describe how pipelined group prefetching works. When the processor executes the PrefetchingLink system call in Figure 2-(a), the operating system sets the P bit of $A{\rightarrow}next$'s memory line and invalidates the line's cached copies. In addition, it adds an entry in the group translation table of the owner network processor. The entry contains the link's line and word number and the virtual and physical address of B. Note that the owner network processor is easily identified with the physical address of $A{\rightarrow}next$. Suppose that, later in the code, a processor misses on A. The request will reach the right memory module, where the network processor will group-prefetch A. While prefetching A, the network processor will check if any P bit is set. If so, the network processor will inspect its table and perform a group-prefetch on the physical address listed in the table.

Similarly, when the L1 Pipelined Prefetching hardware detects that the line that it reads from the secondary cache has its P bit set, it sends a read request for the line. The request will reach the owner network processor which will perform a group prefetch of A and B. Recall that, if the directory in the owner node indicates that the lines are already in the requesting processor's cache, the lines will not be read from memory.

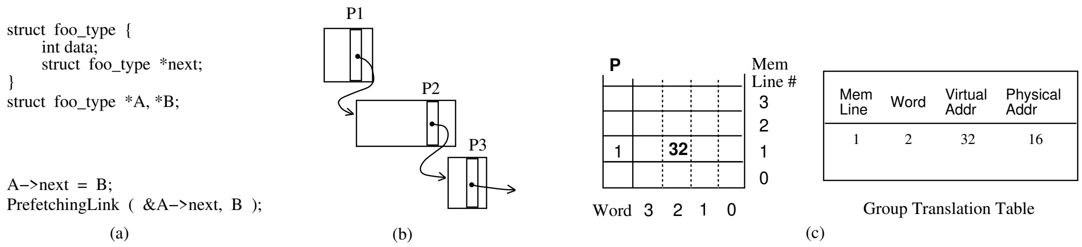


Figure 2: Pipelined prefetching of groups. Chart (a) shows the system call that ties $A \rightarrow next$ to group B. From now on, any prefetch of A also prefetches B. Chart (b) shows several records linked with prefetching links. Finally, Chart (c) shows some of the hardware necessary for pipelined prefetching of groups, namely the P bit and the group translation table in the network processor.

The search in the group translation table should be efficiently done with the use of a hash function. Overall, we have measured the size of this table for the applications and found that, at the most, the table needs to store 1,300 entries. This case occurs in *Pthor*, where 40,000 groups need to be stored among 32 processors. Of course, we may choose to keep a smaller table. If the table overflows, we can simply drop some of the entries. The only effect will be that some groups will not be prefetched.

Finally, group translation tables must be modified when pages containing groups are remapped. This is all done by the page fault handler. First, assume that the page that contains the source group, namely A in Figure 2-(a), is moved to disk and then to another physical address in another memory module. In this case, the entries in the group translation table of the owner network processor that referred to A's prefetching links are moved to disk and then to the new owner network processor.

However, if B's virtual page is unmapped and then mapped in another processor, the work to be done is different. Indeed, when the PrefetchingLink system call in Figure 2-(a) was originally executed, the operating system did an additional operation that was not discussed above. It associated with B's virtual page two pieces of information: B's offset in the page, say *offB*, and the fact that B could be the target of a pipelined prefetching of groups. Suppose that, later, B's physical page is moved to disk and, eventually, the data is reloaded into another physical page. When a processor suffers a page fault trying to access B's page, the operating system, in addition to supplying the new page mapping, it will also inform the faulting processor that *offB* can be the target of a pipelined prefetch. The faulting processor will then search the group translation table in its node and, if it finds an entry for B, it will update the physical address. Conceptually, it is like having to update two page tables.

4.3.1 Applicability of the Optimization

Strictly speaking, this optimization is applicable only to applications whose groups are linked with pointers. This includes *MP3D*, *Barnes*, *Maxflow*, and *Pthor*. For *Barnes* and *Maxflow*, inserting the PrefetchingLink system calls in the right place requires relatively less understanding of the program. Therefore, the programmer can fairly easily give feedback to the compiler. For *MP3D* and *Pthor*, however, the program needs to be understood more. Let us consider the different applications in detail.

In *Maxflow*, the prefetching links are the pointers that link node and edge records to form the graph used in the program. These pointers are set at initialization time and never change. In addition, we also mark as prefetching links the pointers that link edges in an active list. These pointers, however, change at run time several times. Therefore, every time they change, we make a PrefetchingLink system call. Each call, of course, updates the corresponding group translation table. For *Barnes*, the prefetching links are the links between the bodies and cells that form the Barnes tree. These pointers are changed once per time step. For *Pthor*, the prefetching links are the pointers that link the logic gates, called elements, with their input and output nets to form the circuit. In addition, they also include the pointers linking each input net to a dummy entry at the head of its event list. None of these links changes. There are also some other, less important, prefetching links. Finally, for *MP3D*, we manually added a new link between the particle that is currently being processed and the cell of the next particle to process. We mark this link as a prefetching link. This helps prefetch the cell earlier. This is a fairly obvious optimization to perform, and is equivalent to the software pipelining performed by Mowry and Gupta [13] in the same code.

If we are willing to add additional pointers between groups as we did for *MP3D*, the rest of the applications could be easily optimized too. For example, we could have used prefetching links to link the molecules in *Water* and to link matrix columns in *Cholesky* and *Ocean*. These changes would require a moderate understanding of the applications.

4.4 Summary

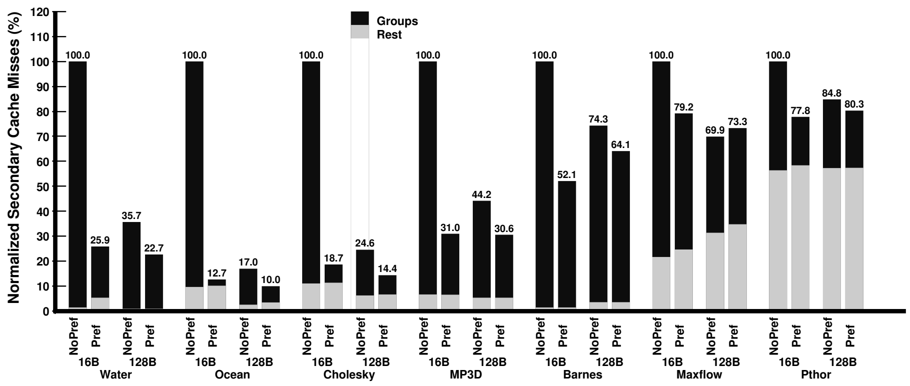
Overall, we propose three optimizations, namely base group prefetching (Section 4.1), L1 pipelined prefetching (Section 4.2), and pipelined prefetching of groups (Section 4.3). The first and third optimizations prefetch the data into the secondary caches; the second one prefetches the data into the primary caches. For these optimizations, we need three bits per memory line, namely P, N, and B. P and N are propagated to the secondary caches, while N is propagated to the primary caches.

5 Evaluation

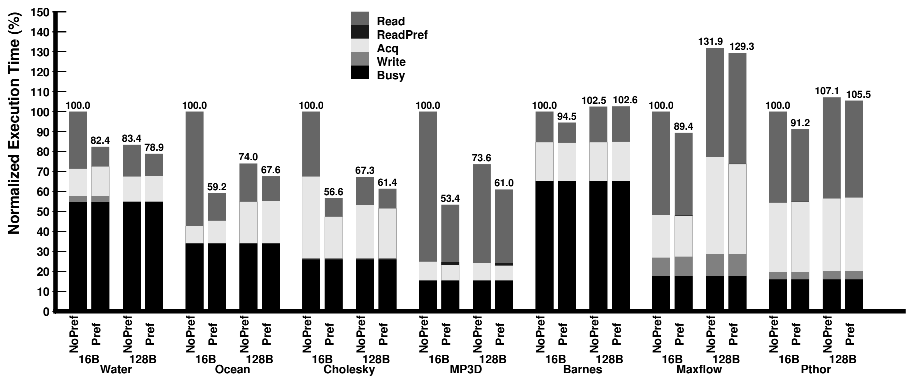
After discussing the support necessary for our optimizations, we now evaluate their performance impact. In the following, we start by describing our evaluation methodology and then evaluate each of the three proposed optimizations in turn.

5.1 Evaluation Methodology

Our evaluation is based on execution-driven simulations of a 32-processor architecture running the parallel applications of Section 3. We use TangoLite [9] to simulate a 32-node NUMA multiprocessor. Each node contains a 200 MHz processor, a 64 Kbyte primary cache, a 256 Kbyte secondary cache, a portion of the global memory with its corresponding directory and a network processor. All caches are direct-mapped. The primary cache is write-through, while the secondary cache is write back and lock-up free. Each node has two 16-entry write buffers: one between the primary and secondary caches, and another between the secondary cache and the node bus. We use the release memory consistency model. The caches use a



(a)



(b)

Figure 3: Read misses in the secondary cache (Chart (a)) and execution time (Chart (b)) with and without base group prefetching. For a given application, the bars are normalized to the number of misses (Chart (a)) or the execution time (Chart (b)) for short cache lines without prefetching.

directory-based cache coherence protocol based on the Illinois protocol. Finally, the nodes are interconnected with a 32-bit wide bidirectional mesh clocked at 200 MHz. All contention in the machine is modeled.

We simulate two different systems: one where memory lines and secondary cache lines are 16 bytes long (*16B*), and one where they are 128 bytes long (*128B*). In both cases, the lines in the primary cache are 16 bytes long. The contention-free latencies in the memory hierarchy of the two systems are shown in Table 2. From the table, we see that it is more costly to service a memory access in the system with long cache lines. In both systems, the *Group* and *PrefetchingLink* system calls execute for 15 cycles in the CPU and for 15 cycles in the network processor.

In our simulations, we only consider shared data because it accounts for the very large majority of the cache misses in these applications. In this paper, we report the number of read misses in different cache configurations relative to the number of read misses in the *16B* system without prefetching optimizations. For the latter system, the absolute read miss rates are as follows. In the primary cache, the miss rates are 1.1% for *Water*, 3.6% for *Ocean*, 3.6% for *Cholesky*, 13.1% for *MP3D*, 6.1% for *Barnes*, 10.1% for *Maxflow*, and 9.2% for *Pthor*. In the secondary cache, the miss rates are 97.8% for *Water*, 81.6% for *Ocean*, 98.4% for *Cholesky*, 97.5% for *MP3D*, 56.4% for *Barnes*, 95.2% for *Maxflow*, and 88.9% for *Pthor*. These large numbers are the result of the primary cache filtering out many references.

Table 2: Contention-free round-trip latencies in cycles for accesses to the different levels of the memory hierarchy.

Memory Hierarchy Level	*16B* (Cycles)	*128B* (Cycles)
Primary Cache	1	1
Secondary Cache	14	14
Memory (Local)	68	96
Memory (Remote, 2 Hops)	132	188
Memory (Remote, 3 Hops)	152	208

5.2 Base Group Prefetching

To evaluate the impact of base group prefetching (Section 4.1), we first examine its effect on the number of secondary cache misses and then its effect on the execution time. Focusing first on the misses, Figure 3-(a) shows, for each application, the number of secondary cache read misses in caches with short lines (leftmost pair of bars) or long lines (rightmost pair of bars). For each cache line size, we show the number misses in the original program (*NoPref*) and after applying base group prefetching (*Pref*). For a given application, the bars are normalized to the number of misses for short lines without group prefetching. For each bar, the misses are divided into misses within groups and the rest of misses. In the figure, memory accesses whose latency is partially hidden by prefetches are not counted as misses.

Focusing first on the *NoPref* bars, we see that, for most of the applications and cache line sizes, the fraction of misses on groups is over 90%. The main exceptions are *Maxflow* and

Pthor, where this fraction falls to 30-75%. Part of the reason why *Pthor* has a small fraction of group misses is that many of *Pthor*'s records are no bigger than a cache line. As a result, if they do not stride two cache lines, they do not qualify as groups. Overall, the many group misses suggest that group prefetching has the potential to make an impact.

If we look at the misses after group prefetching (*Pref* bars), we see that, in most cases, the number of misses decreases significantly. The reduction is mainly caused by fewer misses within groups; the misses outside groups are seldom affected. The reduction, of course, is larger for short cache lines because group prefetching allows them to expose much more spatial locality. For short lines, the miss reduction is about 70-85% for *Water*, *Ocean*, *Cholesky*, and *MP3D*, and 20-50% for the rest. The relatively small miss reduction in *Maxflow* and *Pthor* is partly due to the small size of their groups. For long cache lines, however, reductions are smaller. This is because long lines already help caches exploit some spatial locality. The reduction is 30-40% for *Water*, *Ocean*, *Cholesky* and *MP3D*, and very small or negative for the rest of the applications. The poorer results in *Barnes*, *Maxflow*, and *Pthor* are in part due to the two non-intended effects of prefetching, namely displacement of useful data from the cache and false sharing. Overall, however, group prefetching eliminates many misses with relatively little cost and programmer effort, specially for short cache lines.

To determine the real impact of this optimization, however, we need to examine the change in execution time. This is shown in Figure 3-(b). The figure is laid out like the previous one: for each application, the two leftmost bars correspond to short cache lines, while the two rightmost ones correspond to long lines. For each line size, we show the original execution time (*NoPref*) and the one after group prefetching (*Pref*). For a given application, the bars are normalized to the execution time for short lines without group prefetching. In each bar, the time is divided into processor busy time (*Busy*), stalls due to write buffer overflow and release points (*Write*), processor waiting to acquire a lock (*Acq*), stalls due to read misses partially hidden by prefetches (*ReadPref*), and stalls due to read misses on non-prefetched data (*Read*). The overhead of the group-binding system call is added to *Busy*.

From the figure, we see that, without group prefetching, three applications are slower with long cache lines. These applications are the most irregular ones, namely *Barnes*, *Maxflow*, and *Pthor*. The reason for this effect is that, although caches with long lines suffer fewer misses, each miss is more costly. *Maxflow*, in addition, suffers an increase in synchronization time. We interpret this effect as the result of the higher traffic induced by long cache lines. Such traffic, possibly concentrated in short periods, slows down the passing of locks among processors. For the other four applications, however, the systems with longer cache lines are 20-30% faster.

Focusing now on the impact of prefetching, we examine short lines first. Group prefetching speeds up most of the applications significantly. The actual speedups vary depending on the number of secondary cache misses eliminated and the original secondary and primary cache miss rates. *Water*, *Ocean*, *Cholesky*, and *MP3D* run 20-50% faster, while *Barnes*, *Maxflow*, and *Pthor* run 5-10% faster. The reason for the low speedup of *Barnes* is the small amount of *Read* time to start with. The low speedup of *Maxflow* and *Pthor* is due to the relatively small miss reduction that group prefetching obtains (Figure 3-(a)). We also note that, for *Cholesky*, prefetching causes a reduction in acquire time. This is because processors execute critical sections faster and, therefore, locks pass from one processor to the next faster.

In the systems with long lines, however, the speedups are smaller: *Water*, *Ocean*, *Cholesky*, and *MP3D* speed up by about 5-15%, while the others change very little. The reasons for the lower impact on systems with long lines are the lower miss reductions obtained and the fact that these systems are less tolerant to increases in traffic as induced by group prefetching.

Overall, we conclude that base group prefetching combined with short lines is effective: six applications are now faster with short lines, often by a large amount (15-30%), while one application is only slightly faster with long lines. We also note in passing that, given that the amount of *Busy* time is not affected by the prefetching, the overhead of the group-binding system calls is negligible.

5.3 L1 Pipelined Prefetching

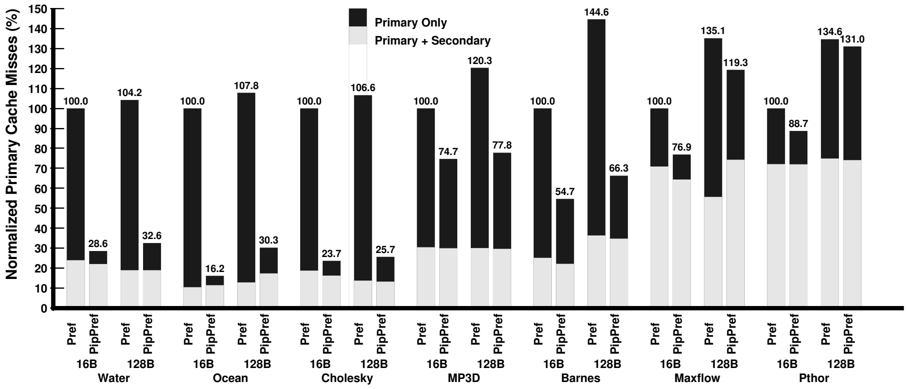
We now augment the *Pref* system with support for L1 pipelined prefetching. The resulting system we call *PipPref*. In the following, we first examine the impact on the primary cache misses, and then the impact on the execution time.

The impact of L1 pipelined prefetching on primary cache read misses is shown in Figure 4-(a). For each application, the leftmost set of two bars corresponds to short cache lines, while the rightmost set corresponds to long lines in the secondary cache. Within a set, from left to right, the bars correspond to the *Pref* and *PipPref* schemes. For a given application, the bars are normalized to the number of misses for short lines in the secondary cache and *Pref*. For each bar, the misses are divided into those suffered by both primary and secondary caches and those suffered by the primary cache only. The former are caused by cold references, data sharing and conflicts in both primary and secondary caches, while the latter are caused by conflicts in the primary cache only. In the figure, memory accesses whose latency is partially hidden by prefetches are not counted as misses.

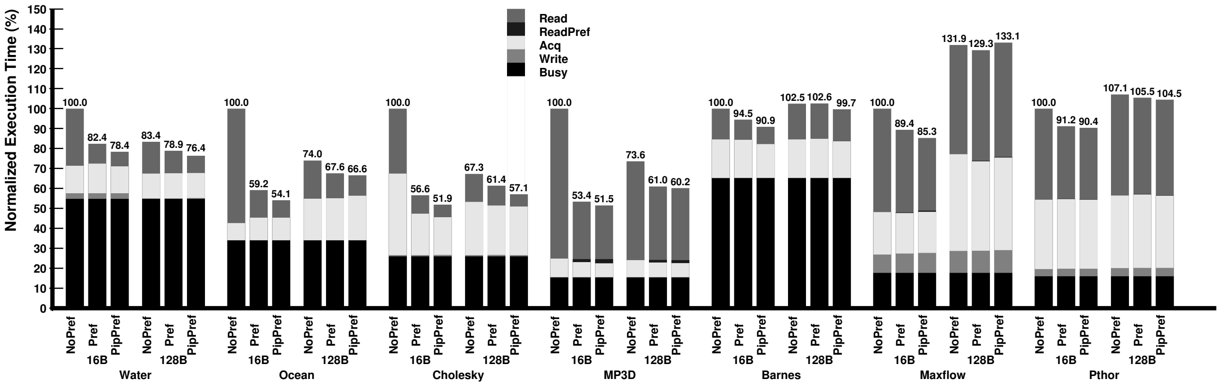
Without L1 Pipelined Prefetching, there are fewer misses in the *16B* system than in the *128B* system. This is because more cache conflicts and false sharing in the secondary cache of the *128B* system cause more invalidations in the primary cache. The *Primary+Secondary* misses, however, seldom increase. This is due to the good prefetching effects caused by long secondary cache lines in the *128B* system. If we now move to the *PipPref* bars, we see that *16B* still has fewer misses. Indeed, focusing first on the *16B* system, we observe that prefetching reduces the number of misses by as much as 45-85% in *Water*, *Ocean*, *Cholesky*, and *Barnes*. For the other applications, the reductions are a more modest 10-25%. The large reductions in *Primary-Only* misses indicate that L1 pipelined prefetching is effective. This scheme works best if the groups are large and their cache lines are accessed in sequence. This is what occurs in *Water*, *Ocean*, *Cholesky*, and *Barnes*. However, two effects can reduce the effectiveness of the scheme. First, pipelining may not work well because either the groups are too small, or not all their cache lines are accessed, or their cache lines are not accessed in sequential order. Second, pipelining may work but, because the working set of the application is large, lines brought in by the prefetcher displace other useful lines from the primary cache too early. A combination of both effects occurs in *MP3D*. For *Maxflow* and *Pthor*, the number of *Primary-Only* misses was small to start with.

Prefetching is also very effective for the *128B* system. It reduces the number of misses in *Water*, *Ocean*, *Cholesky*, and *Barnes* by as much as 55-75%. For the other three applications, the reductions are smaller. In the case of *MP3D*, the relatively small change is due to the same reasons as indicated for short lines. For *Maxflow* and *Pthor*, however, pipelining does not work as well as in *16B* even though the prefetcher still prefetches only 16 bytes at a time from the secondary cache. The cause of the problem is that a secondary cache line with group data now has a higher chance of containing non-group data as well. Since there is only one set of N and B bits per line, the prefetcher cannot distinguish between group and non-group data in the same line. As a result, useless data is brought into the cache, causing displacements of useful data.

To see how the miss reduction translates into speedups, Fig-



(a)



(b)

Figure 4: Read misses in the primary cache (Chart (a)) and execution time (Chart (b)) with and without L1 pipelined prefetching. For a given application, the bars are normalized to the number of misses for *Pref* (Chart (a)) or the execution time for *NoPref* (Chart (b)) and short cache lines.

ure 4-(b) shows Figure 3-(b) plus a *PipPref* bar per setup. The *PipPref* optimization is relatively effective with short cache lines for the three regular applications plus *Maxflow*. Except for *Maxflow*, these applications had the largest miss reductions in Figure 4-(a). Figure 4-(b) shows that they are sped up by 5-9% over *Pref*. The speedups result from reduced *Read* and, sometimes, *Acq* time. For the other applications, the changes in execution time tend to be smaller. For long cache lines, the changes are usually smaller or even negative. The negative change in *Maxflow* is caused by the traffic of useless data between secondary and primary caches. Such traffic slows down the transfer of useful data.

Overall, while *PipPref* does not speed up some of the applications much, it is attractive because it is cheap and it speeds up regular applications for short cache lines. In addition, this optimization will become more effective as the latency gap between primary and secondary caches increases.

5.4 Pipelined Prefetching of Groups

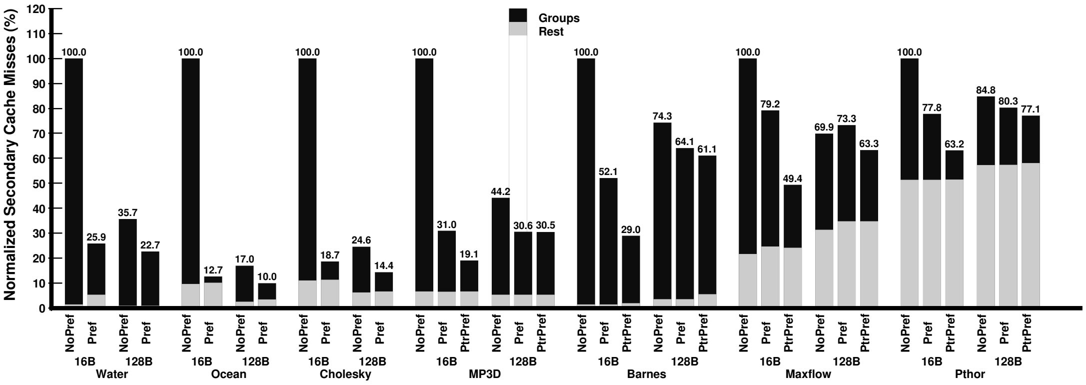
Finally, we augment the *PipPref* system with support for pipelined prefetching of groups into the secondary cache (Section 4.3). The resulting system we call *PtrPref*. This optimization is directly applicable only to the pointer-intensive applications, namely *MP3D*, *Barnes*, *Maxflow*, and *Pthor*. In the following, we first examine the impact on the secondary cache misses, and then on the execution time.

The impact of this optimization on secondary cache read misses is shown in Figure 5-(a). The figure shows Figure 3-(a) plus a *PtrPref* bar per setup in the four rightmost applications. As usual, memory accesses whose latency is partially hidden by prefetches are not counted as misses. In the figure, if we first examine the bars for short cache lines, we see that *PtrPref* reduces the number of group misses by approximately half in all applications. As a result, the total number of misses is reduced by 20-45% over *Pref*. Overall, the data shows that following prefetching links between groups is a good heuristic.

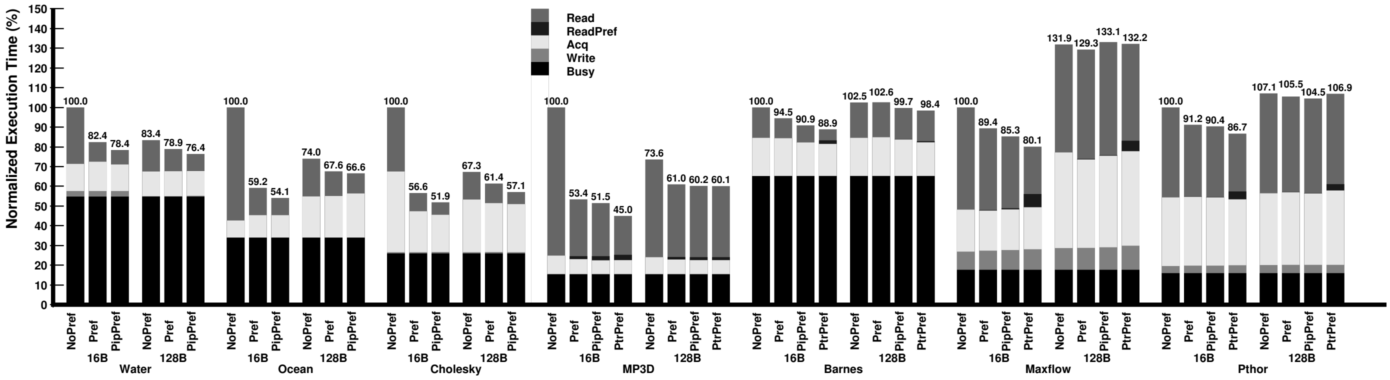
For long cache lines, however, the picture is different. Compared to *Pref*, the misses are reduced by an average of 6% only. The reason is that applications suffer false sharing and extra cache conflicts induced by the transfer of a large amount of data with long cache lines.

To see the impact on the execution time, Figure 5-(b) shows Figure 4-(b) plus a *PtrPref* bar per setup in each of the four pointer-intensive applications. Focusing first on short cache lines, we see that, for *MP3D*, *Maxflow*, and *Pthor*, the optimization reduces the execution time by, on average, 8%. For the other application, namely *Barnes*, the speedup is only 2% because the *Read* time accounts for very little. Overall, while 8% is a relatively good speedup, the impact of this optimization is limited by the ability of the hardware to issue the prefetches early enough. This problem appears when all that the processor does is to follow a chain of pointers. The extent of this problem is shown by the presence of *ReadPref* time in the *Maxflow* and *Pthor* bars of Figure 5-(b). In applications where the processor did some other work while following a chain of pointers, the *ReadPref* time could be decreased and the program sped up more.

For long lines, however, the optimization has practically no effect or a negative one. The reason is the small miss reduction in *MP3D*, *Barnes*, and *Pthor*, and the increase in traffic in *Maxflow*. As an aside, we also note that, since the amount of *Busy* time is not affected by the prefetching, the



(a)



(b)

Figure 5: Read misses in the secondary cache (Chart (a)) and execution time (Chart (b)) with and without pipelined prefetching of groups. For a given application, the bars are normalized to the number of misses (Chart (a)) or the execution time (Chart (b)) for *NoPref* with short lines.

overhead of the *PrefetchingLink* system calls is negligible.

We have manually added links in the data structures of the other three applications as indicated in Section 4.3.1 and performed pipelined prefetching of groups. However, since the size of groups is usually quite big, the additional gains are small. In addition, in *Ocean* and *Cholesky*, there are very few group misses after the base group prefetching has been applied. For *Water*, the addition of prefetching links among molecules eliminates 50% of the remaining secondary cache misses. However, since the *Read* time is so small in Figure 5-(b), the execution time decreases by only 3%.

Overall, pipelined prefetching of groups, while requiring more architectural support than the previous two optimizations, is able to further speed up the most irregular applications. With the three proposed optimizations, systems with short cache lines now run the applications on average 17% faster than systems with long cache lines. The differences are larger for the more irregular applications, where they reach, in one case, 40%.

5.5 Discussion

Our evaluation has pointed out two problems with long cache lines. The first one is that irregular applications slow down if caches have long lines. While in two (*Barnes* and *Pthor*) out of the three most irregular applications we use (*Barnes*, *Maxflow*, and *Pthor*) this slowdown is modest, we believe that there are many other applications that would display larger slowdowns for long lines.

The second problem is that, using prefetching, it is harder to optimize a system with long lines than one with short ones. Indeed, prefetching long lines is more likely to cause false sharing, cache conflicts, and much traffic. This is confirmed by our data: the prefetching optimizations have been much more effective for short lines, both relatively and absolutely speaking. For long lines, it has been harder to reduce the *Read* time. In addition, at least partly because of the traffic increase, the *Acq* time has often increased as well.

As a result, a system with short cache lines and the proposed optimizations runs all the applications except one faster than a system with long lines, with or without the optimizations. In some cases, especially for the irregular applications, the machine with short lines runs as much as 25-40% faster. We have performed similar experiments for systems with 32-byte cache lines and found results similar to the 16-byte data presented. Overall, therefore, if the machine is expected to run a mix of regular and irregular applications, our results suggest the use of short cache lines (16-32 bytes) and the proposed optimizations.

While we recommend all three optimizations, they do not all have the same cost-effectiveness. The most cost-effective one is the first one, namely base group prefetching. Indeed, with little architectural and software support, it achieves large speedups in nearly all applications. After this optimization, all applications except one are already faster with short lines. L1 pipelined prefetching is also very cost-effective. While its impact is smaller and it tends to work mostly for regular applications, it is attractive because it is cheap. Finally, pipelined prefetching of groups is a bit less cost-effective. It requires more complicated hardware and it speeds up irregular applications only. It sped up the three irregular applications with substantial *Read* time by 8% on average. This speedup is on top of the speedups from the two previous optimizations. Furthermore, it is realized on applications that can benefit from very few, if any, other techniques. Moreover, we expect this optimization to make a larger impact on applications where the processor performs some work while following a chain of pointers. In that case, the prefetches issued by the proposed hardware will have time to hide more memory latency and the *ReadPref* time will decrease. We are currently examining other irregular applications.

Finally, we expect that our optimizations will perform a bit better in other, arguably more realistic, architectures. In our simulations, we have used primary caches that are somewhat large relative to the working set of some of our applications. In a real machine, there will probably be more pressure on the primary cache. In that case, the L1 pipelined prefetching scheme is likely to work relatively worse for long cache lines. This is because, as indicated in Section 5.3, L1 pipelined prefetching brings into the primary cache more useless data for long secondary cache lines. When space in the primary cache becomes tighter, the effect of this useless data will be worse. A second issue is that, in our simulations, we have used an aggressive network. This has favored long lines because long lines do not take a lot more time than short lines to travel in the network. In a network with less bandwidth, long lines would perform relatively worse.

6 Comparison to Other Prefetching Schemes

There has been much work on data prefetching, both hardware-based [1, 6, 8, 11] and software-based [2, 5, 10, 12, 13, 14]. Typically, hardware-based schemes only work well for data structures that are accessed with a fixed stride. Therefore, they do not do well in irregular applications like some of ours, where many pointers are dereferenced and references have no clear pattern. An interesting alternative is dynamically scheduled processors with speculative execution. In these systems, however, the window of instructions considered tends to be relatively small and, therefore, these systems issue fewer useful prefetches than our system. Furthermore, the hardware can be quite complex.

Software-based prefetching driven by a compiler has been successfully used in regular applications [10, 14]. However, it cannot be easily applied to irregular applications, where it is hard for compilers to analyze the pointers. While advanced pointer analysis together with the use of profile information may eventually provide some gains, it will surely not be easy to successfully use compiler prefetches for irregular applications.

For irregular applications, instead, researchers have looked at software-based prefetching with hand-inserted prefetches [13]. Compared to our scheme, however, hand-inserting prefetches has four disadvantages. The first one is the overhead of the prefetch instructions and the instructions that generate the addresses for the prefetches. In irregular applications, the latter is likely to be large. In our scheme, instead, there is practically no instruction overhead: the few required system calls add negligible overhead. It could be argued, however, that a superscalar processor could hide the overhead of executing many of these instructions. Whether or not this is true needs to be studied in detail. We are currently addressing this issue.

A second disadvantage of hand-inserting prefetches is the bandwidth consumed by the issue of the prefetch messages. Except for the L1 pipelined prefetching scheme, our techniques all involve bringing many memory lines with only one request to the memory system. The resulting "freed-up" bandwidth may be used by non-prefetching accesses.

A third disadvantage of hand-inserting prefetches is that it requires the programmer to understand and follow the code to a greater degree than the scheme presented. Indeed, if the programmer wants to insert the prefetches by hand, he has to closely examine the code and schedule the prefetches at the right distance from the use of the data. Unfortunately, this is challenging because irregular codes can be very hard to understand. In our scheme, while some programmer effort is also required indeed, we feel that the effort is smaller. This is because the programmer only needs to check for groups of data that are accessed together. For example, to choose our groups and prefetching links, we did not use the extensive profiling used in [13]. Furthermore, adding the prefetches takes longer than adding our system calls. We are currently trying to quantify the difference in effort between the two schemes.

Finally, a fourth disadvantage of hand-inserting prefetches is that it may not be very effective when prefetching records linked by pointers. Often, the programmer can only insert the prefetches after the content of the pointer is read. Unfortunately, the processor will dereference the pointer very soon after that, and only a small amount of latency is likely to be hidden. In the scheme presented, instead, the use of pipelined prefetching of groups allows prefetching to occur earlier. Having said that, there is no guarantee that all the miss latency will be hidden, as we have seen in Section 5.4 for *Maxflow* and *Pthor*.

The scheme presented has three disadvantages compared to software prefetching. First, group prefetching requires a miss in order to trigger the prefetching. As a result, it is not possible to eliminate all misses. Second, the scheme presented may bind groups sub-optimally, thereby creating unnecessary traffic. For example, while some parts of a group may be accessed much less frequently than others, the whole group will be prefetched all the time. Finally, a disadvantage of the scheme presented is the hardware support required. We note, however, that we can still use off-the-shelf processors.

We have attempted to compare our scheme to hand-inserting prefetches. An evaluation of the latter, performed by Mowry and Gupta [13], included *MP3D* and *Pthor* among the applications. Unfortunately, it is hard to compare the quantitative results reported by these authors to ours since they simulated 16 processors, used only 1-Kbyte primary and 2-Kbyte secondary caches for lock-up free caches, and simulated a different architecture. Similarly, all relevant data in [15] corresponds 16 processors.

Interestingly, we do not think that hand-inserted prefetching and our scheme are completely exclusive alternatives. Instead, they can be combined and can complement each other. For example, some hand-inserted prefetches can be used to trigger a group prefetch without requiring the initial miss. Similarly, a hand-inserted prefetch can be used to trigger a pipelined prefetching of groups much in advance. We are currently studying these issues.

7 Conclusions

While many parallel applications have good spatial locality and therefore can use long cache lines, important engineering codes in areas like graph problem-solving or CAD often run better with short cache lines. To effectively run both types of applications, the most intuitive approach is to combine short lines with prefetching. In this way, each application can exploit the amount of spatial locality that it has. In addition, prefetching itself works better with short, rather than long cache lines, since the latter cause more false sharing, cache conflicts, and overall traffic. The contribution of this paper is a new prefetching scheme that, while usable by regular applications, is specifically targeted to irregular ones: *Memory Binding* and *Group Prefetching*.

The basic idea is to bind in hardware and prefetch together groups of data that the compiler or programmer suggest are strongly related to each other. We presented three different prefetch optimizations for systems with short cache lines.

Both base group prefetching and L1 pipelined prefetching are quite cost-effective. The former achieves large speedups in nearly all applications, while the latter further speeds up regular applications. Their cost is low. Pipelined prefetching of groups further speeds up irregular applications but entails a higher architectural cost. For irregular applications with substantial read stall time, it achieves speedups of 8% on average. The speedups are limited by the ability to issue prefetches early enough when the processor is traversing chains of pointers. Larger gains are possible for applications where the processor performs some small amount of work while traversing the chains since, in that case, the prefetches may be issued early enough. We are currently examining the issue further. Despite the relatively modest results achieved so far, however, we still recommend this optimization because it eliminates misses in hard-to-optimize applications.

Overall, our prefetching schemes combined with short cache lines result in a memory hierarchy that can be effectively used by both regular and irregular applications. Therefore, if the machine is expected to run a mix of both types of applications, our results suggest the use of short cache lines (16-32 bytes) and our prefetching instead of long lines (128 bytes) with or without our prefetching. The former system runs 6 out of 7 Splash-class applications faster. In particular, some of the most irregular applications run 25-40% faster.

Acknowledgments

We thank the anonymous referees, Lynn Choi, Joel Emer, Jovan Mitrevski, Alain Raynaud, Marc Snir, Edgar Torres, and Pedro Trancoso for their feedback. We also thank Todd Mowry for the tracing system used in this paper.

References

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

[2] D. Callahan, K. Kennedy, and A. Porterfield. Software Prefetching. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.

[3] R. Chandra, K. Gharachorloo, V. Soundararajan, and A. Gupta. Performance Evaluation of Hybrid Hardware and Software Distributed Shared Memory Protocols. In *Proceedings of the 1994 International Conference on Supercomputing*, July 1994.

[4] T. F. Chen and J. L. Baer. A Performance Study of Software and Hardware Data Prefetching Schemes. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 223–232, April 1994.

[5] W. Y. Chen, S. A. Mahlke, P. P. Chang, and W. W. Hwu. Data Access Microarchitectures for Superscalar Processors with Compiler-Assisted Data Prefetching. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, November 1991.

[6] 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.

[7] C. Dubnicki and T. LeBlanc. Adjustable Block Size Coherent Caches. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 170–180, May 1992.

[8] J. W. C. Fu and J. H. Patel. Data Prefetching in Multiprocessor Vector Cache Memories. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 54–63, May 1991.

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

[10] E. H. Gornish, E. D. Granston, and A. V. Veidenbaum. Compiler-Directed Data Prefetching in Multiprocessors with Memory Hierarchies. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 11–15, June 1990.

[11] 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.

[12] A. C. Klaiber and H. M. Levy. Architecture for Software-Controlled Data Prefetching. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 43–63, May 1991.

[13] T. Mowry and A. Gupta. Tolerating Latency through Software-Controlled Prefetching in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[14] T. Mowry, M. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, October 1992.

[15] T. C. Mowry. Tolerating Latency Through Software-Controlled Data Prefetching. Ph.D. Thesis, Stanford University, March 1994.

[16] 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.

[17] J. Torrellas, M. S. Lam, and J. L. Hennessy. False Sharing and Spatial Locality in Multiprocessor Caches. In *IEEE Trans. on Computers*, pages 651–663, June 1994.

[18] W. Weber and A. Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 243–256, April 1989.

[19] S. Woo, J. Singh, and J. Hennessy. The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 219–229, October 1994.