ISSN 1361 - 6161

THE UNIVERSITY
of MANCHESTER

Computer Science
University of Manchester

Aspects of Hardware-Controlled Data Prefetching

Ando Ki
Alan E. Knowles

Department of Computer Science
University of Manchester
Technical Report Series
UMCS-97-8-1

Aspects of Hardware-Controlled Data Prefetching*

Ando Ki
Alan E. Knowles

Department of Computer Science
University of Manchester
Oxford Road, Manchester, UK.
kiando@cs.man.ac.uk
aek@cs.man.ac.uk

August 1997

Recent technical reports issued by the Department of Computer Science, Manchester University, are available by anonymous ftp from ftp.cs.man.ac.uk in the directory pub/TR. The files are stored as PostScript, in compressed form, with the report number as filename. They can also be obtained on WWW via
URL http://www.cs.man.ac.uk/csonly/cstechrep/index.html. Alternatively, all reports are available by post from The Computer Library, Department of Computer Science, The University, Oxford Road, Manchester M13 9PL, UK.

*Refereed by: J. Viv Woods

Abstract

Data prefetching, which issues data fetch requests prior to actual use, is one of the effective techniques to tackle the effects of memory access latency. In this paper, we use two examples of hardware-controlled data prefetching methods, tagged and stride prefetching, to investigate the impact of several system parameters on the performance of data prefetching. The system parameters, related to memory and cache, include memory bandwidth, memory latency, overall cache size, cache line size, set-associative cache, and lockup-free cache. We conduct simulation studies in the context of shared-memory multiprocessors using SPLASH-2 benchmark programs.

The benefits of data prefetching come from reducing cache misses, and result in reducing memory access stall times experienced by the processor, but at the cost of extra network traffic and memory requests. In general, data prefetching is a very attractive approach in hiding memory access latency time without incurring much overhead. Performance enhancement approaches, such as wider memory bandwidth, larger cache size, and set-associative cache, improve not only general performance but also data prefetching performance in terms of reducing load stall times. Longer memory access latency cannot be hidden by a simple hardware-controlled data prefetching. Thus adaptive mechanisms must be employed. Longer cache lines can enhance the performance of data prefetching, but could be harmful when data sharing effects are dominating. We show that lockup-free cache and sufficient network/memory bandwidth are necessary hardware requirements.

Keywords : Hardware-controlled data prefetching, Memory access stall time, Hiding memory access latency time, Lockup-free cache, Shared-memory multiprocessors

1 Introduction

The speed gap between processors and memory has widened in the last few years, and the technology trends indicate that this speed gap is likely to increase in the future [11]. As the speed gap becomes wider, high-performance processors become more sensitive to stalls for memory accesses. The latency of memory accesses is an important issue in the design of a computer system because processors usually stall for the duration of memory requests.

The most fundamental approach to tackling this latency problem is the adoption of a multi-level memory hierarchy. The goal of the memory hierarchy is to bridge the speed gap between high speed processors and memories by introducing faster cache storage between them. The cache is a small but fast memory whose purpose is to hold currently active data, thereby reducing the number of accesses to memory. Although caches reduce the effective memory access time, cache misses still suffer from large memory access latency.

Hiding memory latency is an attractive way to reduce the effect of the speed gap, and prefetching can hide memory latency by bringing data to the upper level of cache hierarchy and close to the processor before it is actually used.

Data prefetching can be classified based on whether it is *binding* or *non-binding* [10]. With a binding prefetch, the value obtained by a later reference is that obtained at the time of the prefetch. On the other hand, a non-binding prefetch brings the data close to the processor, but the data remains visible to the memory coherency activity and is consistent when the processor actually uses it. The former approach has the severe limitation of how far prefetches can be made before their actual use. The reason is because it must make sure that there is no update during the interval between prefetching and reference.

Prefetching can be accomplished in *implicit* or *explicit* ways. When a cache miss results in bringing a cache line into the cache, if a cache line is larger than one word, then some future references might be satisfied within this newly arrived cache line. As a result, prefetching has been made implicitly. The disadvantages are that a larger line takes longer to fill; it increases memory traffic and contributes to cache pollution due to the larger replacement granularity.

On the other hand, prefetching can be done explicitly by issuing prefetching requests. In *hardware (controlled) prefetching*, the hardware alone decides what data to prefetch and when and where to prefetch the data. Therefore, prefetches are handled dynamically without intervention of the compiler or programmer, and there is no need to change code. Hardware-controlled prefetching schemes can be classified according to whether they are based on spatial locality [5, 21] or calculated stride [7, 20]. The former is called *sequential prefetching* and the latter *stride prefetching*. With *software (directed) prefetching*, the compiler or programmer directs prefetching by inserting prefetch instructions into the code [3, 9, 19].

In this work, we only consider the non-binding and explicit prefetching schemes, and concentrate on hardware-controlled prefetching. Especially, we study Smith's *tagged prefetching* [21] and Sklenar's *stride prefetching* [20]. We address the effects of prefetching in shared-memory multiprocessors. Caches are an integral part of data prefetching, and, therefore, cache parameters, such as overall cache size, cache line size, and cache support, are likely to affect data prefetching performance. Other system parameters such as memory bandwidth and latency also deserve attention to study the effect on data prefetching performance.

The goal of our work is to investigate the impact of cache and system parameters on the performance of prefetching. Our experiments simulate parallel benchmark programs from SPLASH-2 [23] on a shared-memory multiprocessor, with and without hardware-controlled prefetching. The metrics of interest include the reduction of memory stall time and cache misses, the effectiveness of prefetching, and the increase in network traffic.

The remainder of this paper is organised as follows. The following section presents the baseline architecture, the architectural assumptions, the performance metrics, and the benchmark programs used in this work. In section 3, we present the effects of prefetching on the baseline architecture in general. Simulation results for the various aspects with different system parameters are presented in section 4. Finally, we summarise in section 5.

2 Simulation environment

This section presents the baseline architecture, the architectural assumptions, the simulation environment, the performance metrics, and the benchmark programs.

2.1 Baseline architecture

The baseline architecture models a typical shared-memory multiprocessor which consists of a number of processing nodes and shared memory modules interconnected by a network.

Each processing node consists of a processor, a primary cache, and a secondary cache. The processor models a load-store RISC architecture called MUSIC (Manchester University Simplified Instruction Code) processor [13], and it uses a simple stall-on-miss model. All instructions execute in one processor cycle except load and store instructions which require memory references.

For simplicity, in accordance with other work in this area, we assume that the primary cache is a write-through, direct-mapped data cache with no-allocation of blocks on write misses. The secondary cache is a write-back, direct-mapped data cache with write-allocate on write misses. The size of primary cache is smaller than that of secondary cache, and we use a mechanism to guarantee the *content multi-level inclusion property* [1]: the contents of the primary cache are a subset of that of the secondary cache.

The global shared memory modules are organised as low-order interleaved memory, so that multiple memory accesses can be handled concurrently. Data is distributed subpage-wise (subpage size is secondary cache line size) in a round-robin fashion. Each memory module can service a request at every memory cycle.

The network is used for data transfer and works in a pipelined fashion where, each network request suffers a network latency (called network delay or network cycle), but the network can handle a request at every processor cycle. We do not want to make this work too specific to any particular network system design details. However, the data transfer protocol in [16] or [2] can be an alternative choice.

Table 1 shows simulation parameters and the latency times for a memory request without conflict. We simulate all system activities at the level of processor cycles and thus all simulation timing parameters (*e.g.* cache delay, network delay, and memory delay) are expressed in terms of processor cycles. Upon a secondary cache miss, for example, a request takes $\mathcal{T}_{C_p} + \mathcal{T}_{C_s} + \mathcal{T}_N + \mathcal{T}_M$ processor cycles.

2.2 Architectural assumptions

We have a number of architectural assumptions and simplifications in our simulation. We assume that there is sufficient depth of network and memory queues, so that no retries are needed but the effects of network contention and memory conflict are reflected in the queueing delay. It is assumed that the sizes of the secondary cache line and the network transfer unit are the same. Each cache miss, regardless of the cache line size, can be serviced by one network access and one shared memory access.

Table 1: Latency and simulation parameters of the baseline architecture

N_p	8	number of processors
M_i	8	number of memory modules
$\mathcal{T}_{C_p}$	0-cycle	primary cache delay
$\mathcal{T}_{C_s}$	10-cycle	secondary cache delay
$\mathcal{T}_N$	50-cycle	network delay (network cycle)
$\mathcal{T}_M$	50-cycle	memory delay (memory cycle)
L_{C_p}	4-word	primary cache line size
L_{C_s}	8-word	secondary cache line size (subpage)
S_{C_p}	4K-word	direct mapped primary cache size
S_{C_s}	16K-word	direct mapped secondary cache size
S_P	256-word	page size

We assume an ideal instruction cache so that instructions always hit in the primary cache. For simplicity, we only consider global data which are potentially sharable. All local data, such as local variables which are maintained in the processor private stack, always hit in the cache. We assume a write-buffer between the primary cache and the secondary cache, and write hits on the secondary cache do not stall the processor. Write misses on the secondary cache make the processor stall, and loads cannot bypass store in the write-buffer. The caches are *lockup-free* [17] : *i.e.* they can support multiple outstanding requests. When a request misses in the cache, a cache line is reserved and the request is sent to the memory.

Prefetching is considered for load operation only. Prefetched data is brought into the secondary cache, and this data remains visible to the cache coherence protocol to guarantee the non-binding property [19]. We assume that cache lines for prefetches are allocated in the cache at issue time in order to find the interference produced by prefetching.

Prefetches for non-allocated pages are not allowed in order to reduce not-used-prefetches (see subsection 2.4) and to prevent prefetching of out of scope data. The prefetch operation used in this work has the following properties [19]: i) *non-binding property* – prefetch operation does not bring data into a register, ii) *non-blocking property* – prefetch operation does not cause processor stalls, iii) *non-excepting property* – prefetch operation does not make memory access related exceptions, such as bus error and page fault.

2.3 Compiler

The compiler is implemented using Princeton University Lcc [6]. The compiler transforms a C-language source program to an assembly language output which is a simple non-pipelined RISC-like load/store order code [13]. The simulator interprets the assembly language directly. The *word* size is fixed at the host machine's integer type size (typically, it is 32-bit (4-Byte)). The simulator compilation route implements all C-language basic data types, such as *char*, *int*, *float* and *double*, but they are all mapped on to a *word*. This was done for simplicity.

2.4 Performance metrics

The main performance metrics used for this study are *memory access stall time*, *cache miss rate*, *network traffic*, and *prefetch utilisation*. We simulate all system activities at the level of processor cycles, and thus all simulation parameters (*e.g.* cache delay, network delay, and memory delay) and results (*e.g.* execution time and stall time) are expressed in terms of processor cycles.

- The *memory access stall time* is defined as the total stall time experienced by all processors due to memory references divided by the number of processors. The memory access stall time consists of *load stall time* and *store stall time*. These are divided further into *miss stall time* and *pending stall time*. The *miss stall time* comes from waiting for blocks that miss in the cache. The *pending stall time* comes from waiting for blocks that have already been requested by previously issued prefetches, but are not yet complete.

- The *miss rate* is defined as the fraction of global accesses that miss on the secondary cache. The miss rate is divided further into *compulsory misses* or *cold-start misses* (the first reference results in a cache miss), *capacity misses* or *replacement misses* (due to replacement of data by subsequent cache loads), and *coherence misses* (misses which occur as a result of invalidation to preserve multiprocessor cache consistency).

- The *network traffic* is defined as the total number of network activities. We subdivide the network traffic further into *load traffic*, *store traffic*, *coherency traffic* which includes invalidation and write back, *synchronisation traffic*, and *prefetching traffic*.

- The *prefetch utilisation* shows the usage of the prefetched data. The *prefetch efficiency* is defined as the proportion of prefetched data actually used by the processor. There are some cases which can affect the prefetch utilisation. One is *not-used-prefetch* : the prefetched data is never used by processor. The other is *short-lived-prefetch* : the prefetched data is replaced due to lake of cache space or invalidated by another processor before it is used. The former is called *replaced-prefetch* and the latter is *invalidated-prefetch*.

- The *execution time* is defined as the total time spent for all activities that occur during the simulated execution of an application. Furthermore, we subdivide the execution time into *code cycles* (instruction execution time), *memory access stall time* (waiting for data returning), and *synch wait cycles* (waiting for synchronisation).

2.5 Benchmark programs

We use parallel benchmark programs taken from SPLASH-2 [23]. They are written in the C-language [12] with a shared-memory architecture in mind and parallelism is implemented by augmenting the program with the notation of the Argonne National Laboratory parallel macros (PARMACS) [18]. Table 2 summarises benchmark programs.

Table 2: Benchmark Programs

programs	**problem size**	**description**
`barnes`	2K bodies 3 steps	a hierarchical n-body problem using the Barnes-Hut method
`cholesky`	1806×1806 (*tk14*) sparse matrix	a sparse matrix factorisation
`fft`	16K points	a complex 1-D fast Fourier transform using the radix-$\sqrt{n}$ six-step method
`fmm`	1K bodies 5 steps	a hierarchical n-body problem using an adaptive Fast Multipole Method
`lu`	256×256 matrix	LU dense-matrix factorisation (non_contiguous_blocks version)
`ocean`	130×130 grid tolerance 10^{-5}	an ocean basin simulator using a hierarchical multi-grid method (non_contiguous_partitions version)
`radix`	256K items	radix sorting
`water`	343 molecules 3 steps	a water molecular dynamics (water-nsquared version)

The problem size of each benchmark has been kept relatively small because of the slow instruction interpretation rate of the simulator. The problem size limits the amount of parallelism available, which in turn determines the maximum number of processors that can be utilised. To overcome this deficiency, we only consider the statistics gathered from parallel sections of code and configurations with up to 8 processors.

3 Prefetching in general

Prefetching brings data close to the processor before it is actually used, and allows memory accesses to be overlapped with computation and other memory accesses. Ideally, the memory latency could be hidden completely if all data are brought into the data cache before they are used. However, there are many limitations:

- The processor can suffer from pending stalls when the processor needs prefetch data before it has arrived. In which case, the time interval between the prefetch request and the use of the data is smaller than the memory access latency.

Even if the prefetching is issued sufficiently in advance of the use of the data, there are still the following limitations:

- There is a problem of cache replacement. If prefetching attempts to replace useful data from cache, this could cause extra cache misses. Also, potentially useful prefetched data could be replaced before it is used.

- There is a problem of sharing. If prefetched data is actively shared by other processors, then it could be invalidated before it is used. Furthermore, the prefetching can cause unnecessary amounts of false sharing. The false sharing occurs when processors in a shared-memory parallel system make references to different data items within the same coherence block (*e.g.* cache line or block) and unnecessary coherence operations occur.

- There is a problem of network traffic. Some prefetching may attempt to fetch unnecessary data, or some of the prefetched data may be replaced or invalidated before they are used, and therefore some of the prefetching activities generate extra network traffic. Some prefetching of shared data can cause extra invalidation traffic on this data that might yet have to be used exclusively by another processor.

We use representatives of two hardware-controlled prefetching methods for our experiment. One is Smith's *tagged prefetching* [21] and the other is Sklenar's *stride prefetching* [20]. The former method initiates a prefetching request for the next cache line when a demand for a cache line is the first access since a prefetch or miss on it. There is a *tag* bit in each cache line. The tag bit is cleared when its cache line is brought into the cache by miss or prefetch, and marked when its cache line is referenced by the processor. When the tag bit changes to marked, a prefetching request for the next cache line is initiated.

The second method issues a prefetching request when consecutive accesses, generated by the same instruction, have the same stride. It uses a hardware lookup table which stores the instruction and operand addresses previously referenced. When the processor executes a load instruction, the instruction and operand addresses are stored in the table. When a later access sees there is the same instruction address in the table, a stride is calculated by subtracting the old operand address from the new operand address, and the address to be prefetched is calculated by adding the stride to the new operand address. In this work, we use 256 entries for the hardware lookup table. According to the work of Fu *et al.* [7], over 90% of the memory references cause hits in the hardware lookup table with 256 entries.

Results for the baseline architecture (see Table 1) are shown in Figure 1, 2, 3, 4 and 5. All data in these figures are normalised to the result of no-prefetching, excepting load miss rates and prefetch utilisation. Several observations can be made from these results.

- Prefetch can improve performance by reducing program execution time, as shown in Figure 1. Each bar consists of five sections that, from the bottom to the top, correspond to code execution time, load miss stall time, load pending stall time, store miss stall time, and synchronisation time (see subsection 2.4). Store and synchronisation times occupy a large portion.

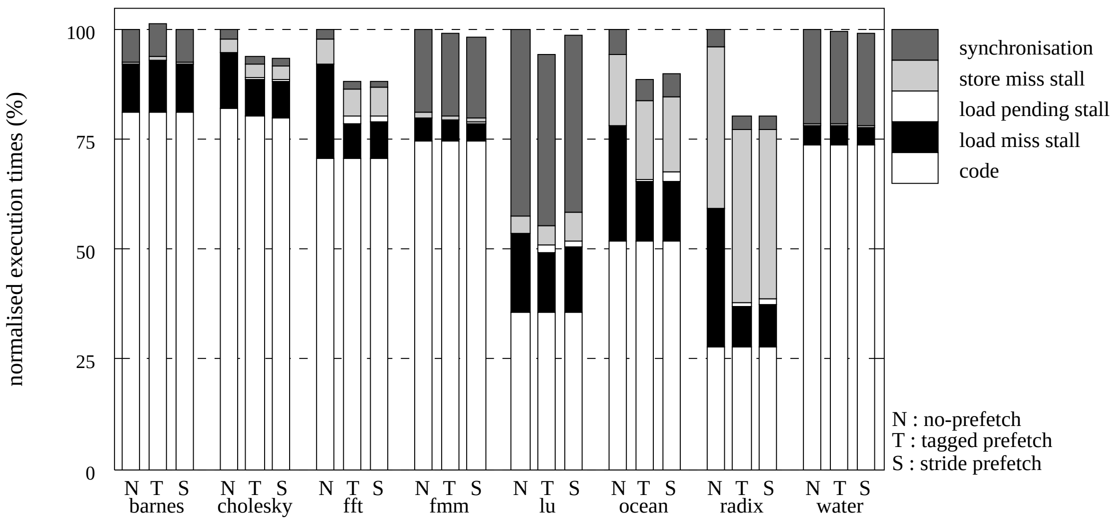

Figure 1: Execution times with baseline architecture: The results are normalised to the case of no-prefetching.

However, we are not interested in those portions. Prefetchings are prohibited during synchronisation, because prefetching for synchronisation variables could make the effects of sharing worse. The store stall times can be removed by employing a write-buffer and a relaxed consistency model [8, 14], so we draw only load stall times in Figure 2 to emphasise the effects on load stall time.

The `cholesky` benchmark uses a task-queue and each process waits on it until new task shows up. This busy-wait is counted as normal instruction execution, hence the number of executed instructions can vary if processes finish their task faster or slower as compare to the original execution.

- Figure 2 shows load stall times which are subdivided into load pending stall time and load miss stall time (see subsection 2.4). Obviously, there are no load pending stall times when there is no prefetching. Load stall times have been reduced by up to 70% by using prefetching.

  Tagged and stride prefetching show nearly identical performance in reducing load stall time for most benchmarks. The main reason is that most benchmarks used in this study have strides shorter than the cache line size. The `barnes` case shows a large stall time with tagged prefetching, and this peculiarity is explained below. Prefetching is not effective if memory stall times are not large due to already high cache hit rates (see Figure 3), as in the cases of `fmm`, `lu`, and `water`.

- Figure 3 depicts the secondary cache load miss rate, which is subdivided into replacement, coherence, and cold-start misses (see subsection 2.4). Load miss rates fell with prefetching, for most benchmark programs except `barnes`. The reason is that prefetching can bring data to the cache before it is actually used.

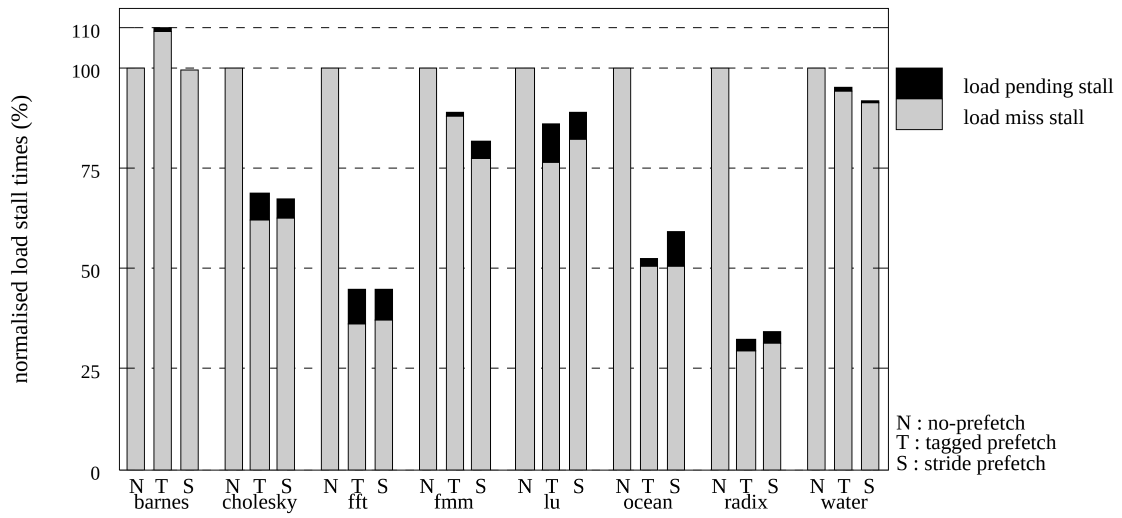

Figure 2: Load stall time with baseline architecture: The results are normalised to the case of no-prefetching.

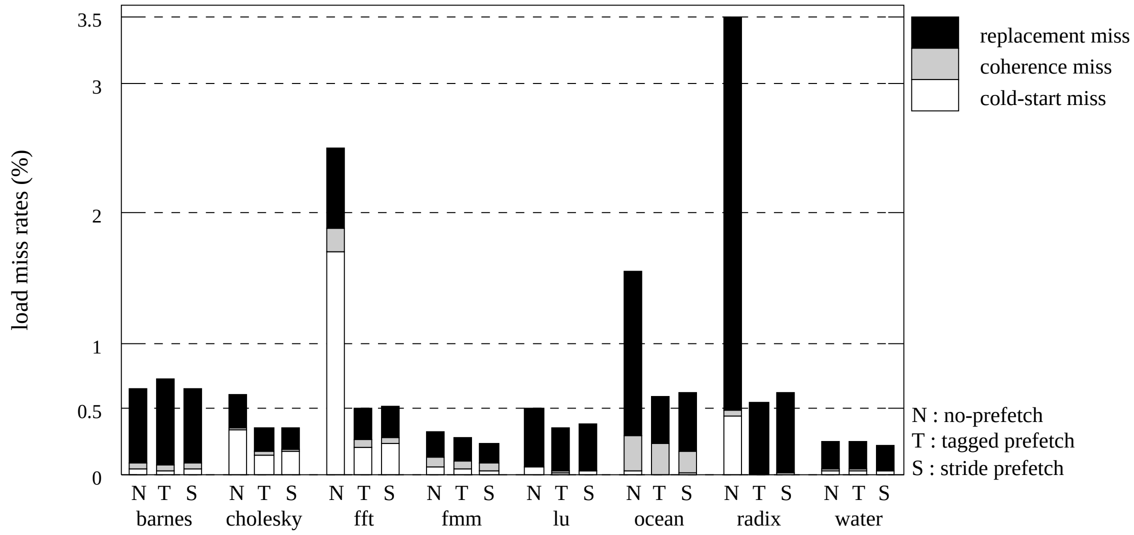

Figure 3: Load misses on secondary cache with baseline architecture (100 $\times$ global_load_misses_on_secondary_cache / global_loads)

The case of `barnes` with tagged prefetching results in more replacement misses, so it suffers from a greater stall time. The reason is that prefetching can cause more replacement by bringing data prior to actual use: *i.e.* prefetched data can replace the data which is actively used by the processor.

In most cases where cold-start misses dominate, cold-start misses are removed by prefetching, as shown in the `fft` and `radix` cases. In spite of the possibility of making extra replacement and coherence misses, most benchmark programs show a considerable reduction of cache misses.

- Figure 4 shows prefetch utilisation which is subdivided into used-, not-used-, replaced-, and invalidated-prefetch (see subsection 2.4). The case of `barnes` with tagged

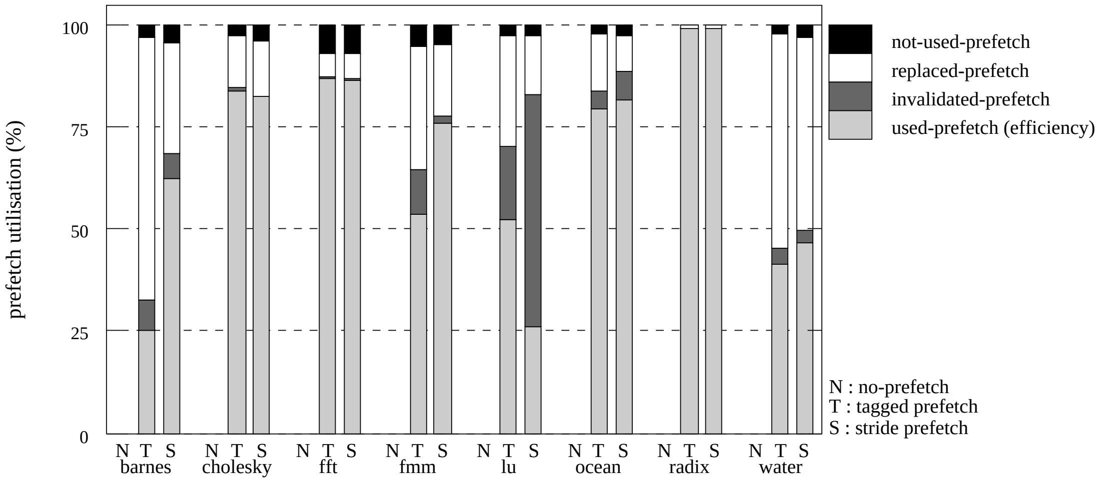

Figure 4: Utilisation of prefetched data with baseline architecture: Bars for no-prefetching are missing because there is no prefetching.

prefetching, and the case of `water` with tagged and stride prefetching show more than 50% of prefetching is replaced-prefetch. As a result, `barnes` and `water` do not get much benefit of reduction in stall time for the baseline configuration as shown in Figure 2. The case of `radix` suggests that prefetching can be totally effective as all prefetched data are useful.

- Figure 5 presents network traffic which is subdivided into synchronisation, coherence, store, prefetch, and load (see subsection 2.4). Network traffic is increased by prefetching, as expected, in all benchmark programs. Five out of eight programs show extra network traffic in the 20% range. The reason can be seen from Figure 4 which shows how much prefetches are actually used. Therefore, lower prefetch efficiency means more network traffic has been generated by prefetching. Note that the section of used-prefetch indicates prefetch efficiency. Network demands increased by prefetching could degrade performance, and this issue is discussed in subsection 4.1. Prefetching, depending on the program, could waste network and memory bandwidth.

The benefits of prefetching come from reducing cache misses, and result in reducing memory access stall time experienced by processor, but cost extra network/memory bandwidth. In general, prefetching is a very attractive approach in reducing data access stall.

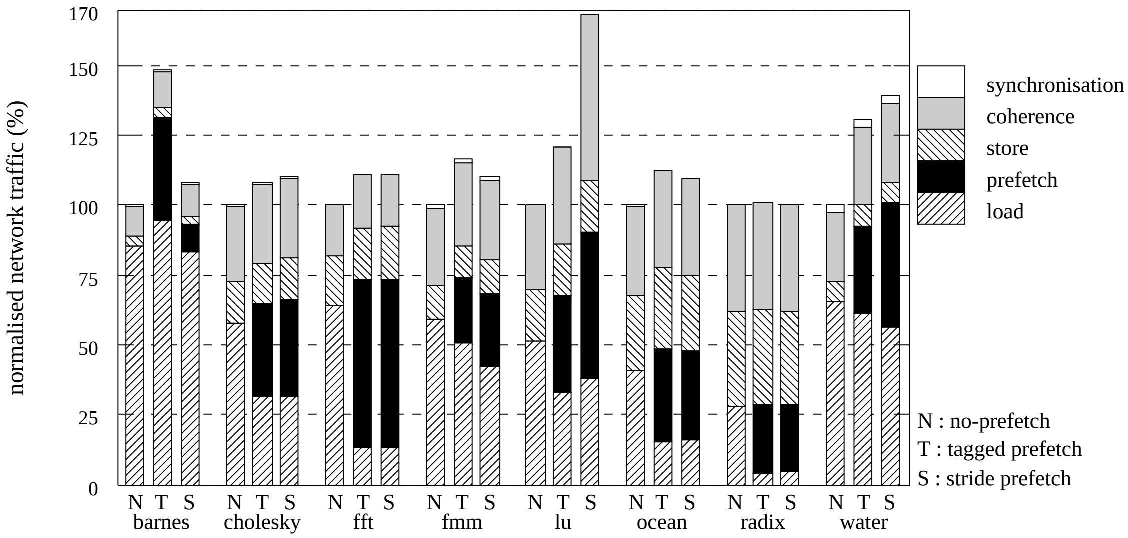

Figure 5: Network traffic with baseline architecture: The results are normalised to the case of no-prefetching.

4 Effects of variation of system parameters

In this section, we evaluate the impact of several system parameters on the performance of prefetching. The variations include changes in the memory configuration, changes in the memory access latency, changes in the sizes of primary and secondary caches and cache lines, changes in set-associativity, and with lockup-free or non-lockup-free cache.

4.1 Effects of memory bandwidth

The multi-way low-order interleaved memory configuration enables processors to access memory modules concurrently. For example, n-way low-order interleaved memory can handle up to n requests unless these requests conflict with each other.

In this subsection, we experiment with five different memory configurations: the one used in the baseline architecture (8-way low-order interleaved memory), 1-way, 2-way, 4-way, 16-way.

In Figure 6, we show the load stall times for these memory configurations normalised with respect to the baseline architecture with no-prefetching and without interleaving (*i.e.* 1-way low-order interleaved memory configuration). In this figure, normalised load stall times, which include miss stall times and pending stall times, versus different memory configurations are depicted. In order to emphasise the variation of pending stall times, we draw them as well.

As could be expected, load stall times increase when memory bandwidth is restricted: *i.e.* stall times decrease as the degree of interleaving increases. As shown in Figure 6, multi-way low-order interleaved memory configurations give better performance by reducing load

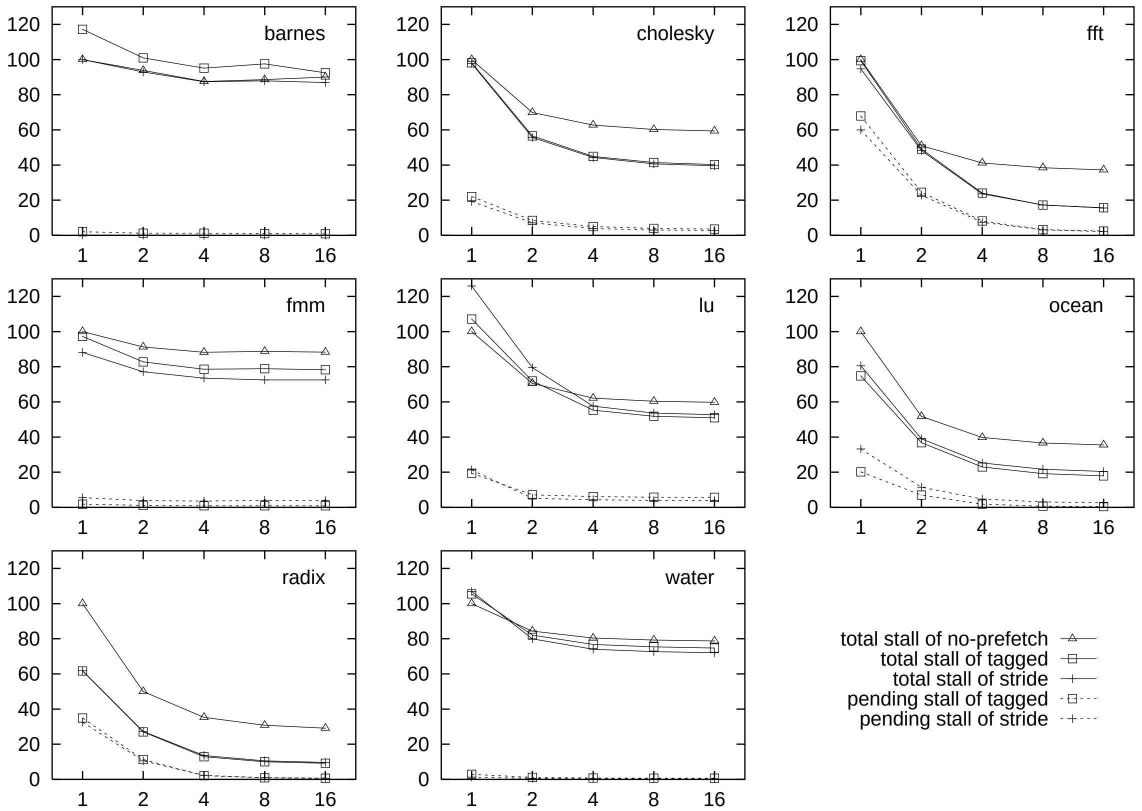

Figure 6: Load stall times (%) with different memory configurations: All data are normalised to the result of no-prefetching without interleaving (*i.e.* 1-way). The figures under the x-axis indicate the degree of low-order interleaving: *i.e.* x-way low-order interleaved memory configuration.

stall times. This improvement comes from low memory contention.

One important phenomenon, that can be observed from the `barnes`, `lu`, and `water` cases in Figure 6, is that prefetching increases stall time rather than decreases it under insufficient memory interleaving: *e.g.* the `lu` case with 1 or 2-way interleaved memory configurations has a greater stall time when stride or tagged prefetching is used (see the `lu` in Figure 6). The reason is that more network/memory requests generated by prefetching are likely to increase network/memory delay due to contention and queueing delay when bandwidth is restricted.

Further observation is that when memory bandwidth is not sufficient, pending stall time increases, as shown in the cases of `cholesky`, `fft`, `lu`, `ocean`, and `radix` in Figure 6. Therefore, to reduce this pending stall time, a lookahead [4] or an adaptive [5, 15] mechanism is needed to issue prefetching requests early. In most cases except `barnes`, prefetching with multi-way interleaving experiences less stall time than no-prefetching.

Prefetching can degrade the performance when the memory bandwidth is not sufficient or memory contention is a primary hindrance. The results presented in this subsection confirm the work of Tullsen and Eggers [22] in showing that an adequate bandwidth is necessary to get the benefits of prefetching.

4.2 Effects of memory access latency

Memory access latency affects directly the effectiveness of data prefetching, because it determines the time spent from the moment the prefetching request is issued, to the moment the prefetched data is returned to the cache. Therefore, as memory latency increases, the number of hits on unfinished-prefetch data can increase. In other words, pending stall time can be a larger portion of stall time when memory access latency increases.

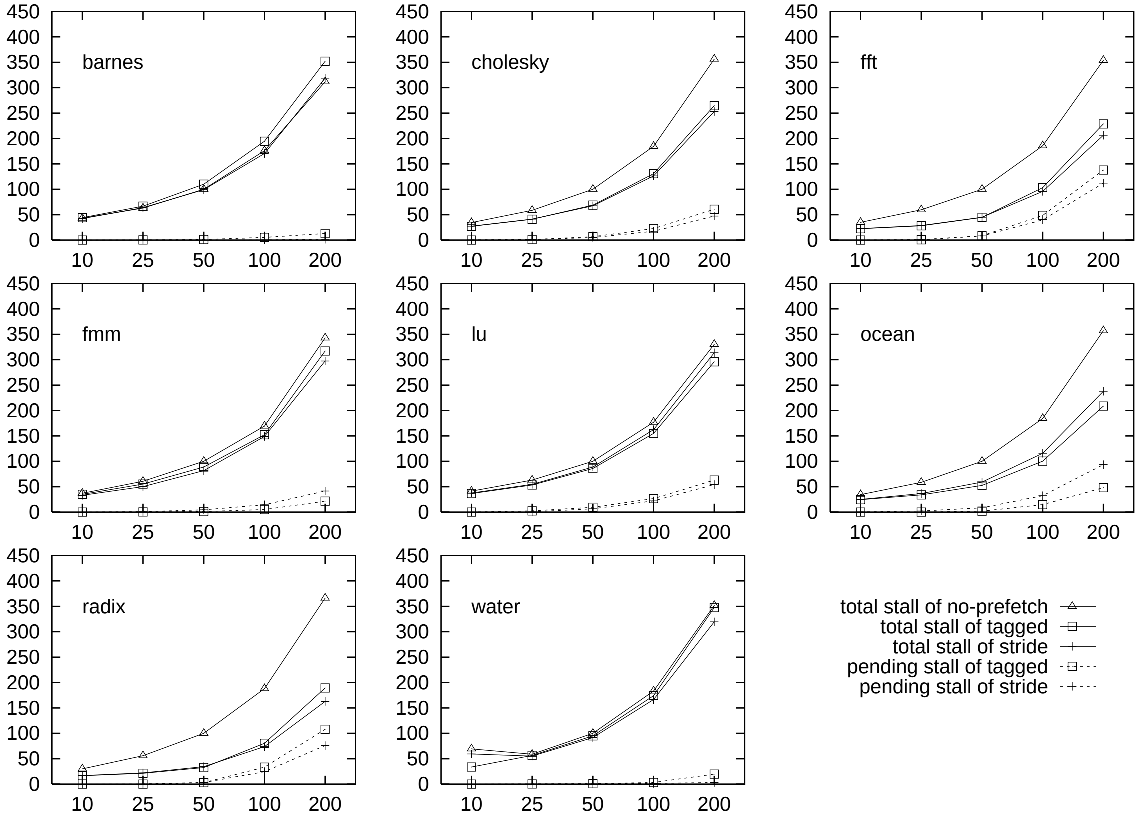

Figure 7: Load stall times (%) with varying memory access latency: All data are normalised to the result of no-prefetching with 50 cycles network delay ($\mathcal{T}_N$) and 50 cycles memory delay ($\mathcal{T}_M$). The figures under the x-axis indicate latencies of the memory ($\mathcal{T}_M$) and the network ($\mathcal{T}_N$).

In this subsection, we examine the impact of five different memory access latencies: the one used in the baseline architecture (50 cycles delay for network and 50 cycles delay for memory), 10, 25, 100, and 200 cycles (*i.e.* the effective memory access latencies upon a secondary cache miss are 100, 20, 50, 200, and 400 cycles).

Figure 7 presents the load stall times with respect to the five memory access latencies. All data are normalised to the result of the baseline configuration with no-prefetching. In this figure, total load stall times and load pending stall times are drawn. Obviously, as latencies increase, stall times increase. As we can see in the load pending stall time curves in Figure 7, as memory access latencies increase, pending stall times increase. This is noticeable in the `fft`, `ocean`, and `radix` cases. Therefore, to get more benefits of prefetching with long memory latency, it is essential to issue a prefetching request early.

Early issued prefetching can eliminate pending stall time if prefetched data arrives before the processor uses it. Prefetching is less useful (and possibly harmful) if there is little latency to hide, as shown in the `barnes` and `water` cases. In the `barnes` case, employing prefetching with long memory latency results in more stall time. In the `water` case, prefetching is less useful.

Prefetching for long latency may not result in completely hiding the latency because it is likely to have significant pending stall time. Therefore, it is necessary to overcome pending stall by employing early prefetching mechanism [4] or adaptive [5, 15] mechanism.

4.3 Effects of cache size

It is generally true that a larger cache has higher cache hit rate than a smaller one, because spatial and temporal localities can be better exploited. Likewise, data prefetching can also benefit from a larger cache, since a larger cache enables prefetched data to reside in the cache for a longer time and to make less cache pollution.

In this subsection, we investigate how variations in cache size influence the performance of prefetching. We consider five cache sizes: the one used in the baseline architecture (4K-word primary cache, 16K-word secondary cache), 1K/4K, 2K/8K, 8K/32K, and 16K/64K pairs. During this experiment, the cache line size is the same as that of the baseline architecture.

It can be said that cache size does not affect pending stall time while network/memory bandwidth is sufficient, because cache size does not alter the memory access latency. However, higher network and memory request rates due to small cache size can affect pending stall time by increasing network and memory contention.

In Figure 8, we show load stall times for the five different cache sizes normalised with respect to the result of the baseline architecture (4K/16K) with no-prefetching. As we can see, the load stall times decrease as cache sizes increase.

The results show that pending stall times are relatively unaffected as cache size changes. However, as shown in the `water` with 1K/4K, tagged prefetching with small cache can degrade performance by making cache pollution worse.

Prefetching reduces load stall times as compared to no-prefetching with larger cache size, as shown in the cases of `cholesky`, `fft`, `ocean`, and `radix`. These results imply that employing prefetching can be an alternative to increasing cache size. However, for those

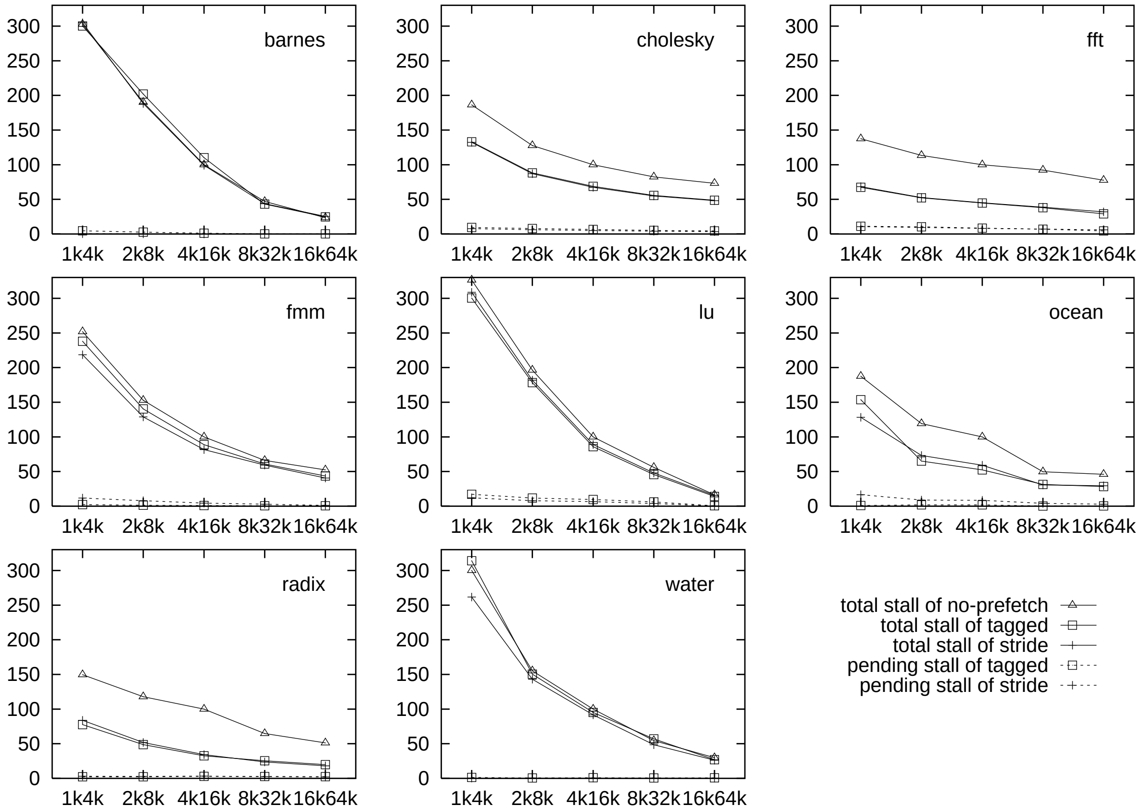

Figure 8: Load stall times (%) with varying cache size: All data are normalised to the result of no-prefetching with 4K-word and 16K-word cache size for the primary and the secondary cache, respectively. The indexes under the x-axis indicate the cache size combinations: akbk (`1k4k, 2k8k, 4k16k, 8k32k, 16k64k`) = aK-word for the primary cache (S_{C_p}) and bK-word for the secondary cache (S_{C_s}).

programs which have high cache hit rates even without prefetching (see Figure 9), such as `barnes`, `fmm`, `lu`, and `water`, doubling cache sizes seems more effective than prefetching.

Figure 9 presents load miss rates on the secondary cache. The main difference with increasing of cache size is reducing replacement miss. Larger cache sizes reduce cache misses, and prefetching with large cache size exhibit an even greater reduction of cache misses.

When memory and network bandwidth are sufficient and data sharing between processors is relatively low, prefetching gets more benefit with a larger cache size.

4.4 Effects of cache line size

Large cache lines can result in implicit prefetching, because the possibility that some future references hit on newly arrived cache line can increase. However, large cache lines can cause

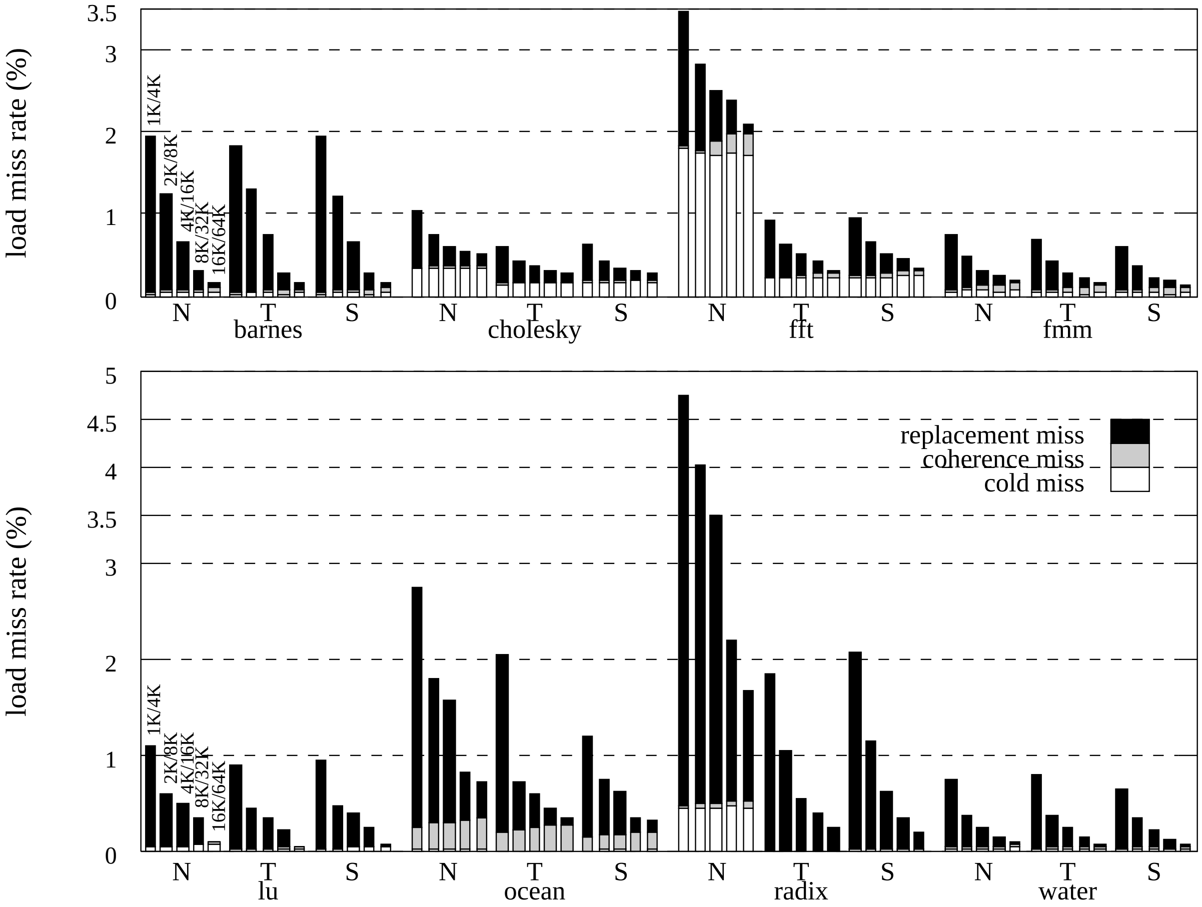

Figure 9: Load miss rates with varying cache size (100 × global_load_misses / global_loads): (N= no-prefetching, T= tagged prefetching, S= stride prefetching) (Each result of prefetchings consists of five bars which represent the results of the primary and secondary cache size combinations: 1K/4K, 2K/8K, 4K/16K, 8K/32K, and 16K/64K)

more conflict misses and false sharing. Therefore, it is not easy to say if prefetching with larger cache line always results in good performance.

In this subsection, we present the effects of cache line sizes with prefetching: 2/4, 4/8 (the one used in the baseline architecture: 4-word for the primary cache line, 8-word for the secondary cache line), 8/16, and 16/32 words pairs. During this experiment, we do not model varying network cost with large cache lines.

There are two groups. One group reduces the load stall times as cache line size increase, in the `cholesky`, `fft`, and `radix` cases of Figure 10. On the other hand, the load stall times increase after decrease, as cache line size increase, as in the `barnes`, `fmm`, `lu`, `ocean`, and `water` cases in Figure 10.

The applications (including `cholesky`, `fft`, and `radix`) having relatively large cold

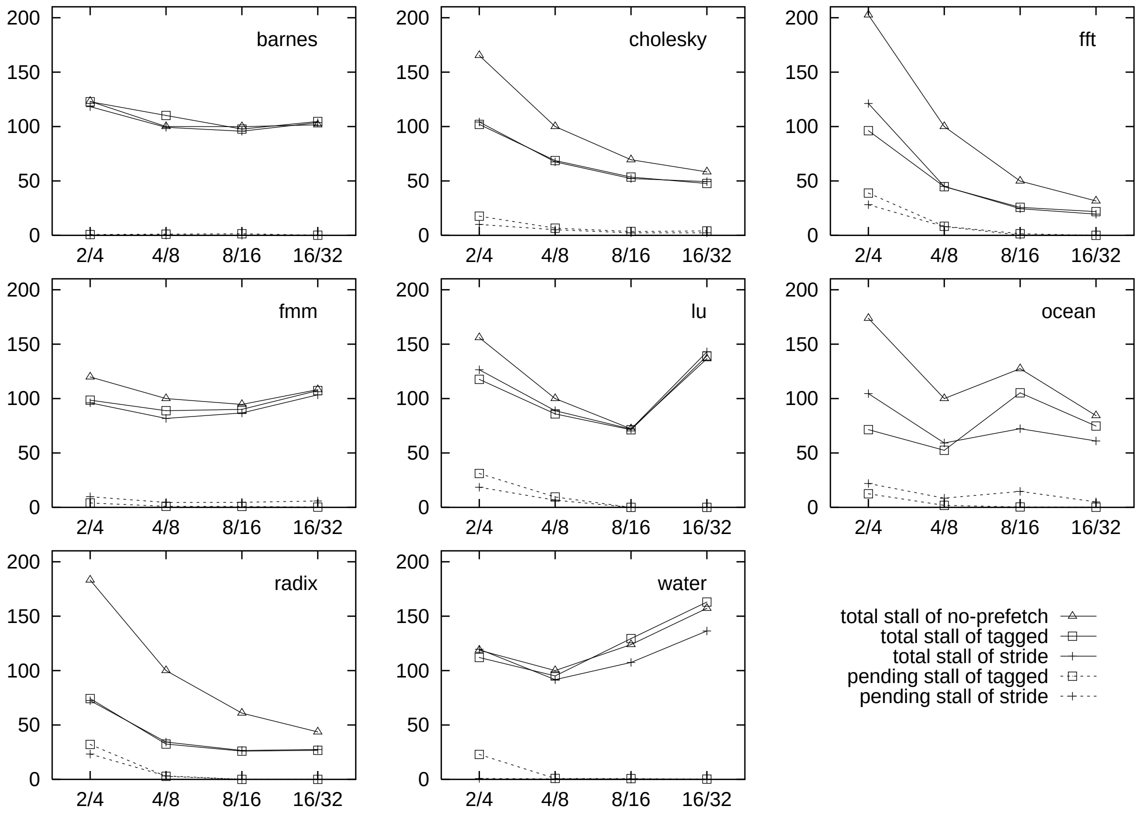

Figure 10: Load stall times (%) with varying cache line size: All data are normalised to the result of no-prefetching with 4-word and 8-word cache line size for the primary and the secondary cache, respectively. The indexes under the x-axis indicate the cache line size combinations: a/b (`2/4, 4/8, 8/16, 16/32`) = a-word for the primary cache line (L_{C_p}) and b-word for the secondary cache line (L_{C_s}).

misses (see Figure 11) get more benefit by increasing cache line size since larger cache lines bring more data. The programs (including `barnes`, `fmm`, and `water`) having high cache hit characteristics lose benefit by increasing cache line size due to increasing replacement misses. The sudden increment of load stall times of the `lu` case at 16/32 cache line size combination in Figure 10 can be explained by looking the coherence miss section in Figure 11. The `lu` case has more coherence misses when cache line size reaches 16/32.

As can be seen in Figure 10, as cache line sizes increase, stall times decrease, and prefetching with large line sizes lead to significant improvements, as the cases of `cholesky`, `fft` and `radix`. The reason is because these benchmark programs show less data sharing characteristics, as shown in coherence misses of Figure 11. As a result, large cache line leads to dramatic reduction of cache misses. In most cases, as cache line sizes increase, the pending stall times decrease. The reason is that larger cache line brings more data and, as

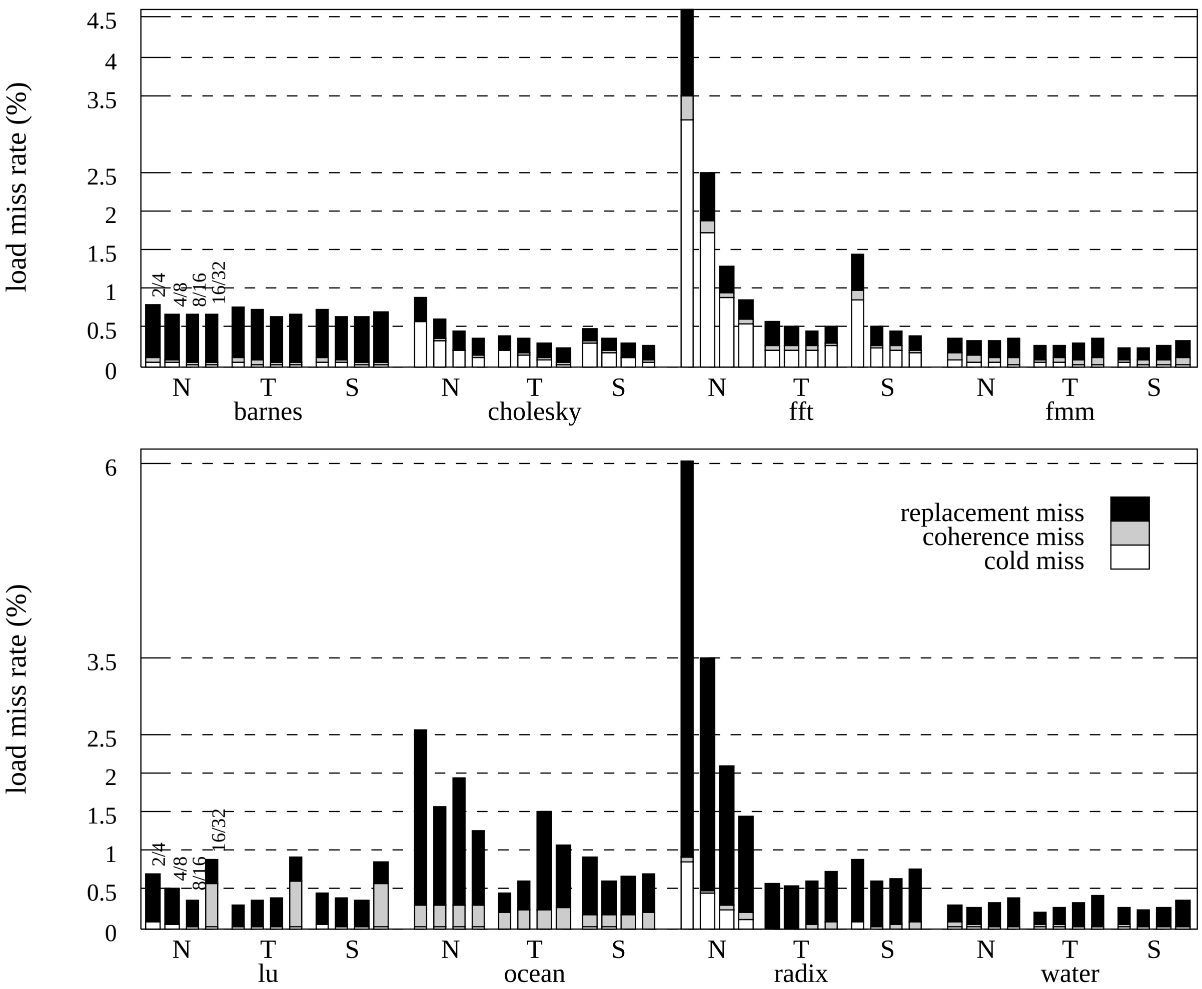

Figure 11: Load miss rates with varying cache line size (100 × global_load_misses / global_loads): (`N`= no-prefetching, `T`= tagged prefetching, `S`= stride prefetching) (Each result of prefetchings consists of four bars which represent the results of the primary and secondary cache line size combinations: 2/4, 4/8, 8/16, and 16/32)

a result, prefetch distance[1] is enlarged.

As expected, invalidation and replacement prefetchings increase with large cache line size, and it results in decrease of prefetch efficiencies, as shown in Figure 12. Note that the used-prefetch section indicates prefetch efficiency.

These results imply that the benefits of prefetching with larger cache line size beyond a certain point are offset by the replacement and coherence misses. Prefetching with larger

[1]The *prefetch distance* is defined by the time period from the moment the prefetching request is issued, to the moment the prefetched data is referred to by the processor.

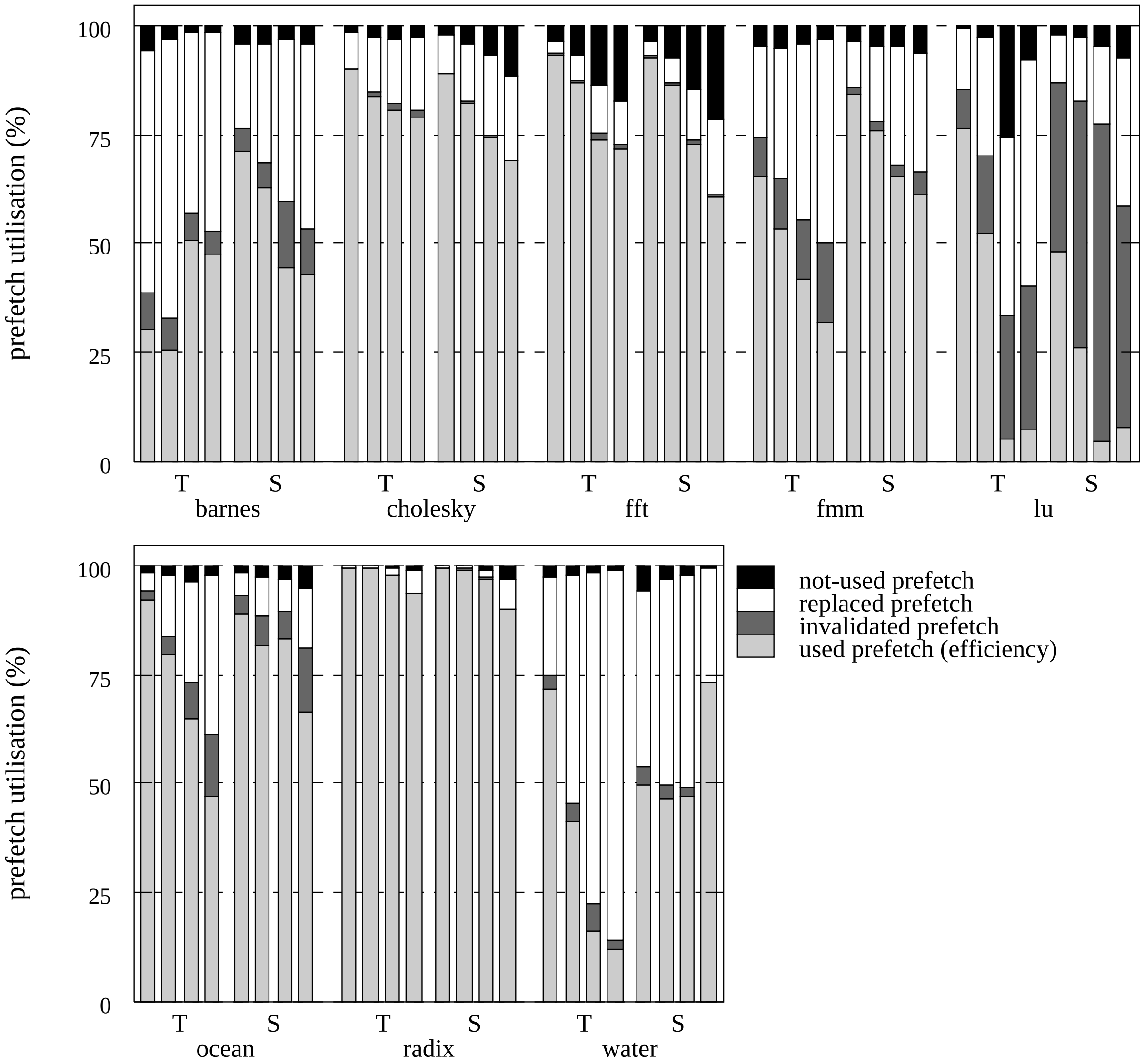

Figure 12: Utilisation of prefetched data with varying cache line size: (T= tagged prefetching, S= stride prefetching) (Each result of prefetchings consists of four bars which represent the results of the primary and secondary cache line size combinations: 2/4, 4/8, 8/16, and 16/32)

cache line sizes is more likely to cause false sharing, cache conflicts, and increased traffic.

Prefetching with a smaller cache line size reduces load stall times as compared to no-prefetching with a larger cache line size, as shown in the `cholesky`, `fft`, `ocean`, and `radix` cases in Figure 10. This result suggests employing prefetching is a viable alternative to using larger cache lines. However, for the cases of `barnes`, `fmm`, `lu`, and `water` which are of high cache utilisation without prefetching, prefetching with a larger cache line size seems not to be beneficial or to even degrade its performance. With the best results presented in Figure 10, a combination of 8-word for the primary cache line size and 16-word for the secondary cache line size seems a good choice to get more benefits with prefetching.

4.5 Effects of set-associative cache

A set associative cache with an appropriate replacement policy, say least-recently-used, usually suffers from less replacement misses because there is an alternative choice for cache line selection and it can exploit temporal locality more effectively. Intuitively, prefetching with a set-associative cache can generate less cache line conflict, and, as a result, data prefetching can give better performance.

In this subsection, we present the effect of three different set-associative degrees of the secondary cache: 1-way (*i.e.* no set-associative or direct-mapped), 2-way, and 4-way set-associative cache.

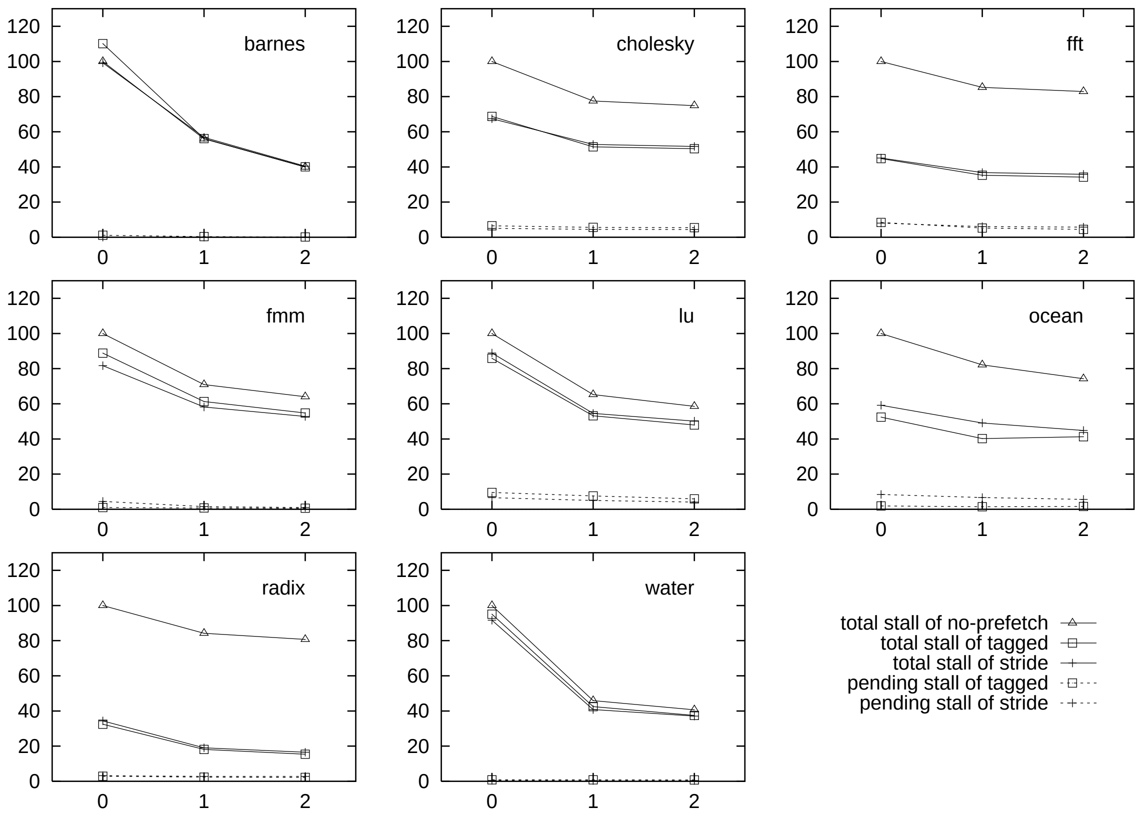

Figure 13: Load stall times (%) with different set-associativities: All data are normalised to the result of no-prefetching with direct-mapped. The figures under the x-axis indicate the degree of set-associativity: 2^x-way set-associativity. Note that 1-way means direct-mapped.

As mentioned at the beginning of this subsection, a higher degree of set-associative reduces the load stall time as shown in Figure 13. Likewise, prefetching also benefits from set-associativity.

4.6 Effects of Cache Support

The secondary caches used in the baseline architecture are lockup-free [17], so that multiple requests can be outstanding. In other words, normal reads are allowed to be serviced while there are prefetching requests outstanding. This ensures that normal reads are not affected by previous prefetchings. Note that the lockup-free cache also allows multiple prefetching requests to be pipelined. Therefore there is no contention on the secondary cache between a processor's request and prefetching, except when both request the same cache line. When caches do not support lockup-free operation, only a single request can be outstanding from the secondary cache at a time.

Non-lockup-free caches reduce stall times less than lockup-free caches, as shown in Figure 14. Since only a single request can be outstanding at a time, the processor's request must wait until the prefetching completes. Especially, in the `barnes` and `water` cases of Figure 14, non-lockup-free cache makes longer load stall times than that of without prefetching. The reason is that outstanding prefetching request makes normal load wait until the prefetching completes. In order to make prefetching contribute to reducing stall time, employing lockup-free cache is essential.

5 Summary

Memory access latency is a well-known obstacle in shared-memory computers. Many solutions have been proposed to deal with this latency problem. In this work, we focus on one of the memory latency hiding techniques, namely non-binding hardware-controlled data prefetching.

The basic idea is to generate data fetch requests for the data cache prior to the actual use of the data. Ideally, the memory latency time could be hidden completely if all data are brought into the data cache before they are used.

The key point of prefetching is deciding when and what to prefetch. Unfortunately, it is difficult to predict what data will be referenced in the future, so that prefetching has limitations and depends on system architecture. If a prediction is wrong, then a cache line is replaced with useless data. If a prefetching is done too early, then it can be replaced or invalidated. These kinds of problems require an extra fetch of the data and increasing coherence and normal traffic.

The benefits of data prefetching come from reducing cache misses, and result in reducing memory access stall times experienced by the processor, but at the cost of extra network traffic and memory requests.

Since prefetching brings data into the cache, caches are an integral part of data prefetching, and cache parameters, such as overall cache size, cache line size, and cache support,

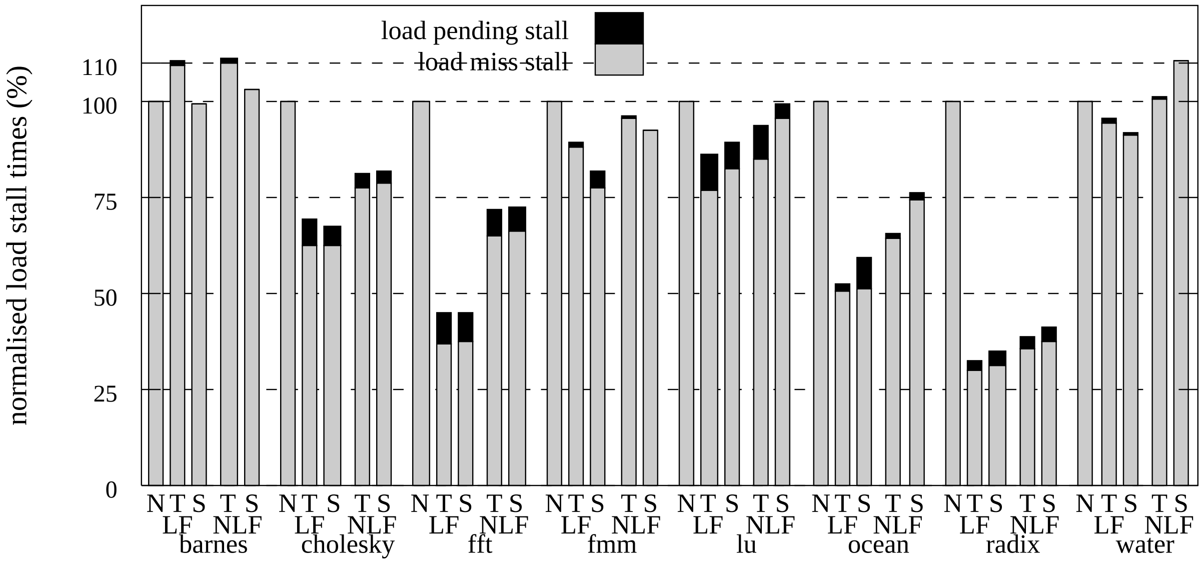

Figure 14: Load stall times with and without lockup-free cache support: All data are normalised to the result of no-prefetching with lockup-free cache. (`N`= no-prefetching, `T`= tagged prefetching, `S`= stride prefetching) (`LF`= with lockup-free cache, `NLF`= with non-lockup-free cache)

are likely to affect data prefetching performance. Other memory parameters, such as memory bandwidth and latency, also deserve attention to study the effect on data prefetching performance.

In this work, we have investigated the impact of several cache and memory parameters, on the performance of prefetching, and presented results from simulation studies in the context of shared-memory multiprocessors using scientific application programs taken from SPLASH-2 benchmark suite.

In order to get benefits of data prefetching, sufficient network and memory bandwidth are necessary, because some extra memory requests are inevitable. Therefore, data prefetching might reduce stall times but it can worsen the performance when the extra requests affect normal memory accesses. These extra requests come from not-used-prefetches whose data are never used by the processor, short-lived-prefetches including replaced-prefetches and invalidated-prefetches, and extra requests due to cache replacements.

Long memory access latency cannot be overcome simply by issuing data prefetching, because there is a high possibility of pending stall times which come from cache hits on unfinished prefetches. The only way to deal with pending stall time is to increase the prefetch distance which is the time interval between prefetch issue and actual reference. Therefore, a lookahead or an adaptive mechanism is necessary to issue data prefetching early.

Larger caches enable both caches with and without data prefetching to reduce stall times. For the applications with relatively higher cache miss rate, prefetching with a smaller cache reduces load stall times as compared to no-prefetching with larger cache size, and it implies that employing prefetching can be an alternative to increasing cache size.

Larger cache lines exploit implicit prefetching due to spatial locality. Likewise, prefetching with larger cache lines can get additional benefits unless false sharing and replacement misses are a serious problem. For these applications, prefetching can be an alternative to increasing the cache line size. Beyond a certain point, prefetching with long cache lines offsets its benefits by incurring false sharing and replacement misses.

The main benefits of data prefetching is bringing data into the cache before the processor needs it. When the processor needs the data which misses on the cache, the normal request must be resolved as soon as possible to minimise stall time. Therefore, the cache must handle more than one request simultaneously because the normal request and prefetching are asynchronous. To do this, prefetching requires a lockup-free cache since prefetching without it could cause the normal request to wait.

In summary, in order to realise benefits of data prefetching, the following system support is necessary: sufficient network and memory bandwidth, lockup-free feature of the cache, and an early-issuing or adaptive data prefetching mechanism. Cache performance enhancing approaches including larger cache size and higher degree of set-associativity, also improve data prefetching performance. The optimum size of cache or cache line should depend on program characteristics. Generally, employing data prefetching instead of increasing cache size or cache line size can be an alternative choice for programs which have relatively high cache miss rates.

Acknowledgements

We would like to thank *Manchester Computing* for providing us access to the *IBM Computer Intensive Facility* which consists of IBM RS6000 computers.
We also thank Dr. J. Viv Woods for helpful comments on the earlier version of the paper.
Ando Ki is supported by ETRI (*Electronics and Telecommunications Research Institute*, the Republic of Korea) and *The British Council*.

References

[1] Jean-Loup Baer and Wen-Hann Wang. Architectural choices for multilevel cache hierarchies. In *International Conference on Parallel Processing*, pages 258–261, August 1987.

[2] Richard R. Billig. A fast backplane cluster heralds a 1000-MIPS computer. *Electronic Design*, July 1987.

[3] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *ASPLOS-IV*, pages 40–52, 1991.

[4] Tien-Fu Chen and Jean-Loup Baer. Reducing memory latency via non-blocking and prefetching caches. In *ASPLOS-V*, pages 51–61, 1992.

[5] Fredrik Dahlgren, Michel Dubois, and Per Stenström. Sequential hardware prefetching in shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(7):733–746, July 1995.

[6] Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin Cummings, 1995.

[7] John W. C. Fu, Janak H. Patel, and Bob L. Janssens. Stride directed prefetching in scalar processors. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 102–110, 1992.

[8] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *ASPLOS-IV*, pages 245–257, 1991.

[9] Edward H. Gornish, Elana D. Granston, and Alexander V. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *International Conference on Supercomputing*, pages 354–368, June 1990.

[10] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd C. Mowry, and Wolf-Dietrich Weber. Comparative evaluation of latency reducing and tolerating techniques. In *18th ISCA*, pages 254–263, 1991.

[11] John L. Hennessy and Norman P. Jouppi. Computer technology and architecture: An evolving interaction. *IEEE Computer*, 24(9):18–29, September 1991.

[12] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, second edition, 1988.

[13] Ando Ki. MUSIC Processor: Manchester university simplified instruction code processor. internal report, University of Manchester, 1995. Department of Computer Science.

[14] Ando Ki. The effectiveness of write-buffers in reducing the impact of memory latency. internal report, University of Manchester, 1997. Department of Computer Science.

[15] Ando Ki and Alan E. Knowles. Adaptive data prefetching using cache information. In *Proceedings of the 11th ACM International Conference on Supercomputing*, pages 204–212. ACM, July 1997. Vienna, Austria.

[16] Ando Ki, Byung Kwan Park, Won Sae Sim, Kyeng Yong Kang, and Yong Ho Yoon. Highly Pipelined Bus : HiPi-Bus. In *Proceedings of the Joint Technical Conference*, pages 528–533, 1991. Hirosima Japan.

[17] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *8th ISCA*, pages 81–87, 1981.

[18] Ewing Lusk, James Boyle, Ralph Bntler, Terrence Disz, Barnett Glickfeld, Ross Overbeek, James Patterson, and Rick Stevens. *Portable Programs for Parallel Processors.* Holt, Rinehar and Winston, Inc., 1987.

[19] Todd C. Mowry and Anoop Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12:87–106, 1991.

[20] Ivan Sklenar. Prefetch unit for vector operations on scalar computers. *Computer Architecture News*, 20(4):31–37, September 1992.

[21] Alan Jay Smith. Sequential program prefetching in memory hierarchies. *IEEE Computer*, 11(2):7–21, December 1978.

[22] Dean M. Tullsen and Susan J. Eggers. Limitations of cache prefetching on a bus-based multiprocessor. In *20th ISCA*, pages 278–288, 1993.

[23] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and methodological considerations. In *22nd ISCA*, pages 24–36, 1995.