ISSN 1361 - 6161

THE UNIVERSITY *of* MANCHESTER

Computer Science
University of Manchester

Secondary Cache Data Prefetching for Multiprocessors

Ando Ki
Alan E. Knowles

Department of Computer Science
University of Manchester

Technical Report Series
UMCS-97-3-1

Secondary Cache Data Prefetching for Multiprocessors*

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

1 March 1997

*Refereed by: Alasdair Rawsthorne

Abstract

We have developed and evaluated a prefetch method that enables stride prefetching at the secondary cache without accessing the processor's internal resources. It uses a data-range-table that enables it to detect usable strides and memory access streams which fall into the same data range. By using program driven simulation of scientific applications in the context of shared-memory multiprocessors, we show that the proposed method can reduce load stall times by an amount comparable to a conventional stride driven prefetching method which requires access to the processor's instruction address register.

Keywords : Data prefetching, Stride, Cache, Multiprocessors

1 Introduction

The speed gap between processors and memories makes it difficult to achieve high processor efficiency, and conventional high-performance computers use a cache memory hierarchy to reduce the effective memory access time. Furthermore, to improve cache utilisation, prefetch techniques have been used.

Prefetching brings data to the upper level of cache hierarchy and close to the processor before it is actually used. The key point of prefetch is deciding when and what to prefetch and therefore the main challenge is predicting what data will be referenced in the future. There are two main hardware approaches : one is *sequential prefetch* whose success depends on spatial locality, and the other is *stride prefetch* that is based on stride calculation.

In the sequential prefetch method, access to a cache line causes prefetching for successive line or lines. If there is sufficient spatial locality, this approach can work well. Methods in the literature [18, 9, 3] can be classified as sequential prefetch methods.

In a stride prefetch method, the instruction and operand addresses of each load are stored in a hardware lookup table, and the instruction address enables the system to identify the access stream which belongs to the same instruction. The operand address of each load is compared with the previous one from the same instruction address and the difference of these two operand addresses becomes the stride. The stride is used to determine the possible next access by adding the stride and the current operand address. Methods in the literature [6, 17, 2] can be classified as stride prefetch methods.

Conventional high-performance computers usually use a two-level cache hierarchy which consists of a fast but small primary on-chip cache and a larger secondary off-chip/on-board cache. Various hardware data prefetching methods have been studied, and prefetching for a secondary cache located outside the processor is quite different from that for an on-chip primary cache. The contents of the program counter are not available to the secondary cache. It is not easy to find the exact instruction address of a given operand address when the processor uses multi-stage pipelines or super-scalar pipelines. Not all access addresses are available to the secondary cache because only addresses generated a result of primary cache misses are visible. Thus, it is not easy to find the exact access streams and predict future accesses.

In this study, we consider a stride prefetch method for secondary cache without modifying the processor or its on-chip cache. This method uses a data-range-table and this enables the hardware to detect a memory access stream whose addresses fall into the same data range. The data range is provided by the programmer, but it can be done automatically by a compiler.

The remainder of this paper is organised as follows. The following section discusses data prefetching for the secondary cache. Section 3 presents an overview of the data-range-table scheme. In section 4, the baseline architecture, the architectural assumptions, the performance metrics and the benchmark programs are presented. Simulation results are presented in section 5. Section 6 describes related works. Finally, conclusions are presented in section 7.

2 Secondary cache data prefetching

Sequential prefetch methods are attractive for the secondary cache because the address of the primary cache miss can be used to generate a prefetch. However, performance improvement gained by this method can be disappointing because the primary cache may satisfy most accesses which have spatial locality and the accesses made to the secondary cache may not have strong spatial locality. Nevertheless, if there is sufficient spatial locality and the primary cache cannot satisfy all requests due to its size limit, and if the stride of a stream is smaller than the secondary cache line size and the addresses of an access stream increase monotonically, sequential prefetch can be an effective approach.

It is not easy to calculate the precise stride outside the processor due to lack of information. If the stride is equal to or larger than the primary cache line size, an accurate stride can be calculated by using the addresses of primary cache misses. Even when the stride is smaller than the primary cache line size, a useful stride can be derived when the addresses of the primary cache misses monotonically increase or decrease. However, the primary cache conflict misses make more misses visible to the secondary cache, and access streams by different instructions may reference the same data. These two cases make it difficult to calculate useful strides.

Let assume a primary cache line size L, and a load instruction within a loop generates an access stream with stride S. If S is equal to or larger than L, then each access of the access stream causes a primary cache miss. Therefore, exact stride can be calculated by subtracting two consecutive miss addresses of the same access stream.

If S is smaller than L, then address differences of two consecutive primary cache misses are some numbers[1] between S and $S \times \lceil \frac{L}{S} \rceil$ inclusively (see appendix A). Therefore if the calculated stride is less than $2 \times L$, prefetching the next cache line is enough to cover smaller stride accesses, because $S \times \lceil \frac{L}{S} \rceil < L + S$ (see appendix B), and $L + S < 2 \times L$ when $S < L$.

No matter what the relationship between S and L, we can use the calculated stride to predict next access. The main point of stride calculation is determining which access stream the primary cache miss belongs to. There have been proposed many complex methods to detect streams (see section 6). In this work, we propose a simple method.

3 Data range table

Miss addresses of the primary cache can be either virtual or real/physical addresses. Therefore, we have to consider both real and virtual address for the secondary cache. Firstly, we assume that the memory management unit (MMU) is available to the data-range-table (DRT), and the DRT receives virtual address and then generates prefetch requests using virtual addresses. Secondly, we consider a DRT that uses physical addresses.

3.1 Using virtual addresses

There are two hardware tables which are *data-range-table* (DRT) and *stride-prediction-table* (SPT), and figure 1 illustrates the logical organisation of these tables. The DRT is a hardware table filled by program.

begin	end	prev_addr	stride	state
o				
o				
o				

DRT SPT

Figure 1: Virtual address data-range-table (DRT) and stride-prediction-table (SPT)

Each entry of the DRT consists of 2 fields :

[1] $\lceil N \rceil$: the least integer greater than or equal to N.

- **begin** : the beginning address of a collection of data structures.
- **end** : the ending address of a collection of data structures.

The SPT stores the addresses previously referenced. For each DRT entry there is a SPT entry consists of 3 fields :

- **prev_addr** : the last address that missed in the primary cache.
- **stride** : the difference between the last two addresses that missed.
- **state** : a valid bit for the entry.

When a reference misses in the primary cache, the address of the reference (miss address) determines which data range the address belongs to. If there is a corresponding DRT entry, *data-range-hit* occurs. Otherwise, *data-range-miss* occurs, and nothing happens. When *data-range-hit* occurs, the corresponding SPT entry is updated as follows:

IF the *state* field is not `valid`,

THEN the *prev_addr* field is set to the primary cache miss address (`Am`), the *stride* field to `0`, and the *state* field to `valid`.

ELSE a stride (`Sc`) is calculated by subtracting the contents of *prev_addr* from `Am`,

    IF `Sc` and the contents of *stride* field are same,
    THEN IF `Sc` is smaller than or equal to the the secondary line size (`Ls`),
        THEN prefetch the next secondary cache line,
        ELSE prefetch the address obtained by adding `Sc` and `Am`.
        ENDIF
    ELSE IF `Sc` is smaller than 2×`Lp`, where `Lp` is the size of primary cache line,
        THEN prefetch the next secondary cache line,
        ELSE prefetch the address obtained by adding `Sc` and `Am`,
        ENDIF
        the *stride* field is set to `Sc`.
    ENDIF

    the *prev_addr* field is set to `Am`.

ENDIF

To fill the DRT, we need some special but simple architectural support including DRT and SPT hardware tables. We must have a way to pass the data range information to the hardware. Data ranges are marked in the source code with the system function *DataRangeSet()*, which takes as arguments the beginning and end addresses of a data set. The DataRangeSet() function fills an entry of DRT with its arguments. Physically, the processor sets the DRT by storing the range data in a memory-mapped location. This filling procedure can be done by a compiler automatically. The compiler computes the starting and ending addresses of a collection of data structures, and generates code to load these into the DRT. This code usually executed only once, and therefore the overhead of filling the DRT table is negligible.

A closer look at the behaviours of the benchmark programs suggests that a DRT of ten to sixteen entries is sufficient since most benchmark programs used have less than ten global data structures.

page_tag	prev_addr	stride	state
∘			
∘			
∘			

DRT SPT

Figure 2: Physical DRT and SPT

3.2 Using physical address

In order to use physical address instead of virtual address, we simply modify the DRT as shown in figure 2. We use higher bits of the physical address for detecting accesses which belong to the same data range. We call these higher address bits the page_tag, which is unique for each physical page. In contrast to virtual address DRT, there is no need for a DataRangeSet() system function to fill the DRT. When page-fault occurs, an entry of the DRT is allocated and its page_tag field is written by page-fault handler. There is another simple way to fill the DRT: when the primary cache miss occurs, check the DRT, if there is no entry corresponding to the page_tag, then allocate one entry, and fill the page_tag. In this work we use a least-recently-used allocation policy and 128 entries. The operation of SPT is same as virtual address DRT. Due to the use of physical addresses, we have to discard the prefetching requests that are beyond a page boundary. This treatment can reduce useless prefetching. The fact is that physically pages may not allocated consecutively, so that beyond page boundary accesses could reference unrelated data.

3.3 Qualitative concern

When each access stream, that is made by an instruction loop, falls into different data range, the DRT method can extract the exact stride of each access stream. However, if there are more than one access stream falling into the same data range, the DRT method cannot calculate the exact stride of a stream. The reason is that there is no mechanism to distinguish between access streams, that fall into the same data range, made by different instructions. There is a solution to this problem. We can split a data range into several sub-data ranges so that each access stream falls into a different sub-data range. However, this approach needs intensive analysis of data access pattern. We do not consider this optimisation with a virtual address DRT in this work.

With physical address DRT method, we could get a benefit when a physical page is divided into more than one partition. This means a physical page can be divided into two or more page_tags. This refinement can help to detect streams that fall into one page. The physical address DRT method has some advantages over the virtual address method. It does not need software assistance and a system function, and can be applicable without data reference analysis. It does not need modification of the cache. However, the physical address DRT method cannot detect a stride that is larger than the physical page size and will not operate across page boundaries.

4 Simulation environment

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

4.1 Baseline architecture

The baseline architecture for our simulation models a typical shared memory multiprocessor which consists of a number of processing nodes and shared memory modules interconnected by a network, as shown in figure 3.

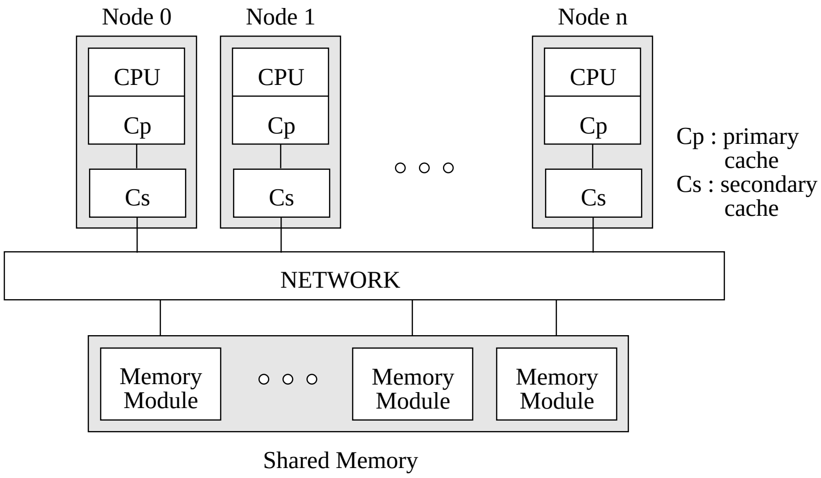

Figure 3: Baseline architecture

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[10]. All instructions execute in one processor cycle except those load and store instructions which require memory references. 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 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 the secondary cache line size) in a round-robin fashion. Each memory module can service a request at a every memory cycle. The network is used for data transfer and it works in a pipelined fashion. Thus, a request can be transfered at every network cycle. We do not want to make this work too specific to any particular network system design details.

Table 1 shows the latency times for a memory request without conflict and other simulation parameters. 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) are expressed in terms of processor cycles. Upon a secondary cache miss, for example, a request takes $T_{C_p} + T_{C_s} + T_N + T_M$ processor cycles. In practice, a word means 4-byte, so that the size of the primary cache is 16K-byte and that of the secondary cache is 64K-byte. If we assume an architectural model in which the processor runs 250-MHz processor cycle and one instruction per cycle, then 10-cycle secondary cache delay results in 40-$n$sec and 50-cycle means 200-$n$sec physical delay. We use relatively small cache sizes in comparing with realistic sizes, and, however, it has been chosen to get realistic cache results, for example a realistic replacement miss rate, because we use relatively small problem sizes for the benchmark programs and we only consider global memory accesses. How prefetching behaves for different architectural parameters has been presented in [11].

We have made some 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 secondary cache line size and network transfer unit are the same, and 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
T_{C_p}	0-cycle	primary cache hit delay
T_{C_s}	10-cycle	secondary cache delay
T_N	50-cycle	network delay (network cycle)
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 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 primary cache. We assume a write-buffer between the primary cache and the secondary cache. Write hits on the secondary cache do not stall the processor, but write misses on the secondary cache make the processor stall, and loads cannot bypass store in the write-buffer. The caches are *lockup-free*[12] : 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 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[15]. 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 and to prevent from prefetching out of data scope of programs. The prefetch operation used in this work has following properties[15]: non-binding, non-blocking property, and non-excepting property.[2]

The compiler is implemented using the Princeton University Lcc[5]. The compiler transforms a C-language source program to an assembler language output which is a simple non-pipelined RISC like "load/store" order code[10]. The simulator interprets the assembler 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.

4.2 Performance metrics

The main performance metrics we use for this study are *load stall time*, *network traffic* and *load miss rate*.

- The *load miss rate* is defined as the fraction of global loads that miss on the secondary cache. The miss rate is divided further into *cold-start miss rate*, *replacement miss rate* and *coherence miss rate*.[3]
- The *load stall time* is defined as the total stall time experienced by all processors due to memory loads of global data divided by the number of processors. The load stall time is divided further

[2]Properties of prefetch operation: *non-binding property*: prefetch operation does not bring data into a register, *non-blocking property*: prefetch operation does not cause processor stalls, *non-excepting property*: prefetch operation does not make memory access related exceptions, such as bus error and page fault.

[3]Cache misses can be classified as following: *Compulsory misses* or *cold-start misses* : The first reference results in a cache miss. *Capacity misses* or *replacement misses* : Misses due to replacement of data by subsequent cache loads. *Coherence misses* : Misses which occur as a result of invalidation to preserve multiprocessor cache consistency.

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.

We concentrate on how to reduce the load stall because write stall can be completely hidden by using write-buffer and relaxed consistency model[7]. Under the release consistency model one of relaxed consistency models, write requests can be overlapped with other memory requests and computations.

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

4.3 Benchmark programs

We use parallel benchmark programs taken from SPLASH-2[19]. They are written in the C-language 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)[13]. Table 2 summarises benchmark programs.

Table 2: Benchmark Programs

	size	**description**
`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 even when running on a IBM Power Series 6000 Model 590 computer which uses the IBM Power 2 RISC chip running at 66MHz with 0.5GByte real memory [14]. 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.

5 Evaluation

In this section, we evaluate the performance of the DRT method. The first set of experiments shows the performance of various prefetching methods: no-prefetching, conventional stride prefetching, virtual-address DRT prefetching and physical-address DRT prefetching. The second set of experiments shows the impact of different page partitions and the number of DRT entries on the performance of physical-address DRT prefetching.

5.1 Comparison with stride prefetching

To measure the effectiveness of a DRT method described in section 3, we use the conventional stride directed prefetching method proposed in [17] (henceforth referred to as *stride method*). In the stride method, we assume that the instruction addresses are somehow available outside the processor. The stride method uses a hardware lookup table called the *stride table* (henceforth referred to as ST) which stores the instruction and operand addresses previously referenced. Each ST entry consists of 4 fields : the instruction address, the operand address, the stride, and a valid bit. When the processor executes a load instruction, an entry of the ST is allocated and the instruction address (program counter) and operand address are stored in the instruction address field and the operand address field of ST entry, respectively. When a later access sees there is the same instruction address in the ST, a stride is calculated by subtracting the old operand address from the new operand address, and then the next possible operand address is calculated by adding the stride and the new operand address. The stride scheme issues a prefetch when two consecutive accesses, which are generated by the same instruction, have same stride. Comparing the DRT and stride method is a useful exercise because the stride method will capture all possible access streams. In this work, we use 256 entries for ST because this size of ST will make 90% hit-rate on ST[6].

Figure 4 presents load stall times with respect to the four different prefetching methods. The load stall time is divided further into *miss stall time* and *pending stall time* (see subsection 4.2). Most benchmark programs experience reduced load stall times by using prefetching, except `fmm` and `lu` with virtual-address DRT. For these programs, virtual-address DRT prefetching shows an increased load stall time, due to low

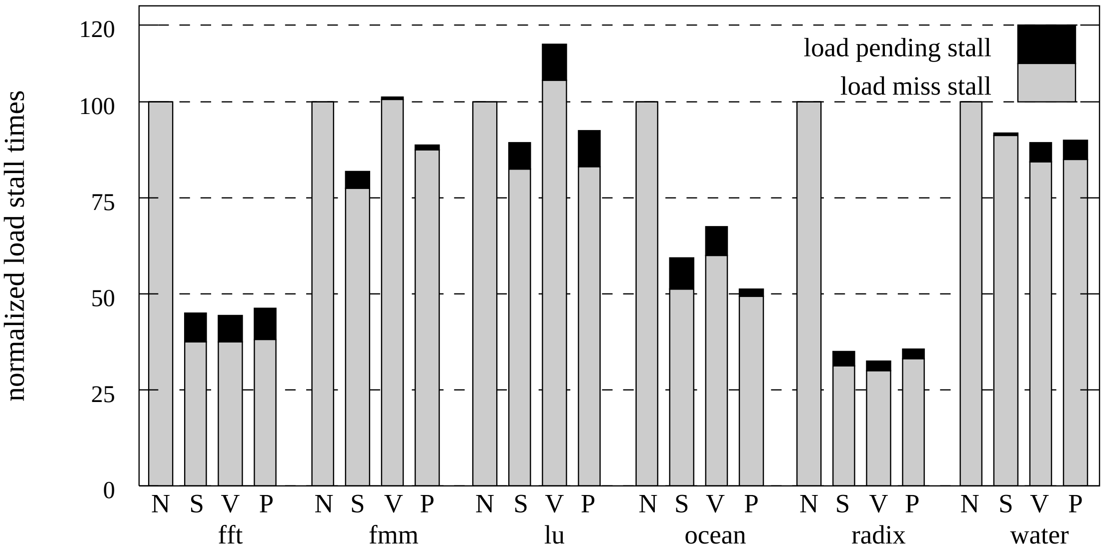

Figure 4: Load stall times with different prefetching schemes: All data are normalised to the result of no-prefetching. (`N`= no-prefetching, `S`= stride prefetching, `V`= virtual address DRT, `P`= physical address DRT)

prefetching efficiency. Figure 6 shows the network traffic and `fmm` and `lu` with DRT prefetching suffer from large extra network traffic. This means that prefetching does not bring the data of normal loads. Most benchmark programs with DRT prefetching get performance improvement comparable to that with stride prefetching. In the `ocean` case, physical-address DRT shows a better performance improvement than stride prefetching. In the `radix` case, virtual-address DRT shows a better performance improvement than stride prefetching.

Figure 5 shows load miss rates with respect to the four different prefetching methods. The load miss rate is divided further into *load cold-start miss rate*, *load replacement miss* and *load coherence miss rate* (see subsection 4.2). Load cache misses decrease when prefetching is employed. Prefetching has little effect on cache miss rate if cache hit rates are already high, as in the cases of `fmm`, `lu` and `water`. As we can see for `fft`, `ocean` and `radix`, load misses decrease dramatically when any sort of prefetching is used. As we can see in `fft` and `radix`, *cold-start misses* and *replacement misses* are reduced dramatically.

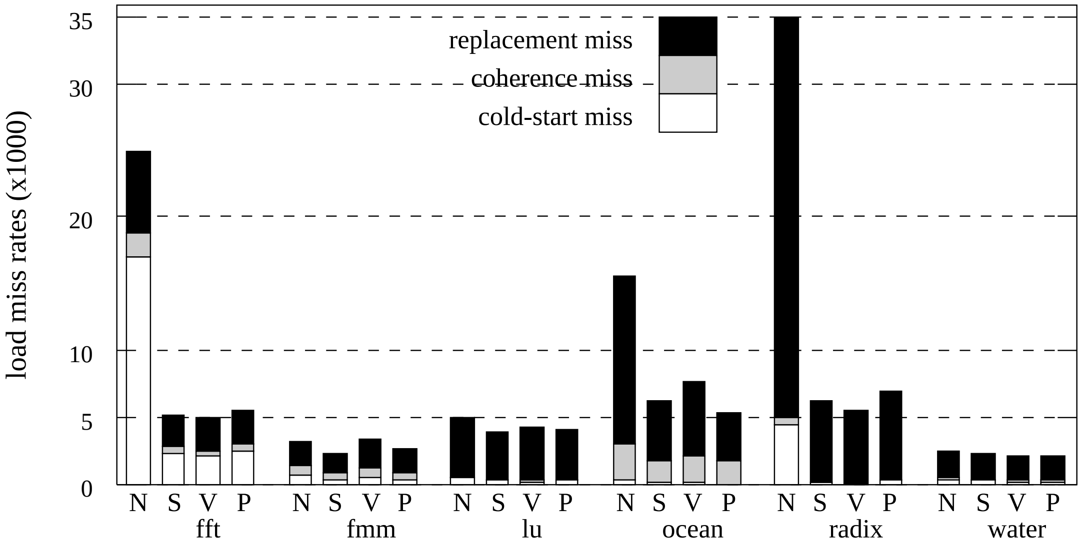

Figure 5: Load miss rates with different prefetching schemes (1000 × global_load_misses_on_secondary_cache / global_loads): (`N`= no-prefetching, `S`= stride prefetching, `V`= virtual address DRT, `P`= physical address DRT)

Most benchmark programs with DRT prefetching yield a reduction of load misses comparable to that obtained with stride prefetching.

Figure 6 shows network traffic with respect to the four different prefetching methods. We subdivide the network traffic further into *load traffic*, *store traffic*, *coherency traffic*, *synchronisation traffic*, and *prefetching traffic* (see subsection 4.2). Network traffic increases, as expected, in all benchmark programs with prefetching. In `fft, ocean` and `radix` cases, a lot of loads are covered by prefetching. As a result, prefetching traffic does not cost extra network bandwidth in `fft, ocean` and `radix`. On the other hand, the `fmm, lu` and `water` cases suffer from extra prefetching traffic that do not cover the normal fetches.

5.2 Performance impact of page partitions

As we mentioned in subsection 3.3, if more than one access stream falls into a physical page, then the physical-address DRT method cannot detect the exact access stream and stride. We may get a benefit when a physical page is divided into more than one partition. This refinement may be helpful to detect the exact access streams and strides when each access stream falls into a different partition.

To show the effect of page partitioning on physical-address DRT method, we experiment with three different schemes. One partition uses a page size of 256 words (1024 bytes). Two partitions use a half page size and four partitions a quarter page size. Therefore there are two partitions within a page when two partitions are used. We do not consider the case that combines more than one page into a partition because physical pages may not allocated consecutively.

As we can see in figure 7, page partitions have little effect on performance. However, more partitions increase stall time slightly, except in the `lu` case. The reason is that increasing the number of partitions causes some access streams to be divided into two or more unconnected streams, and this can lose prefetching opportunities.

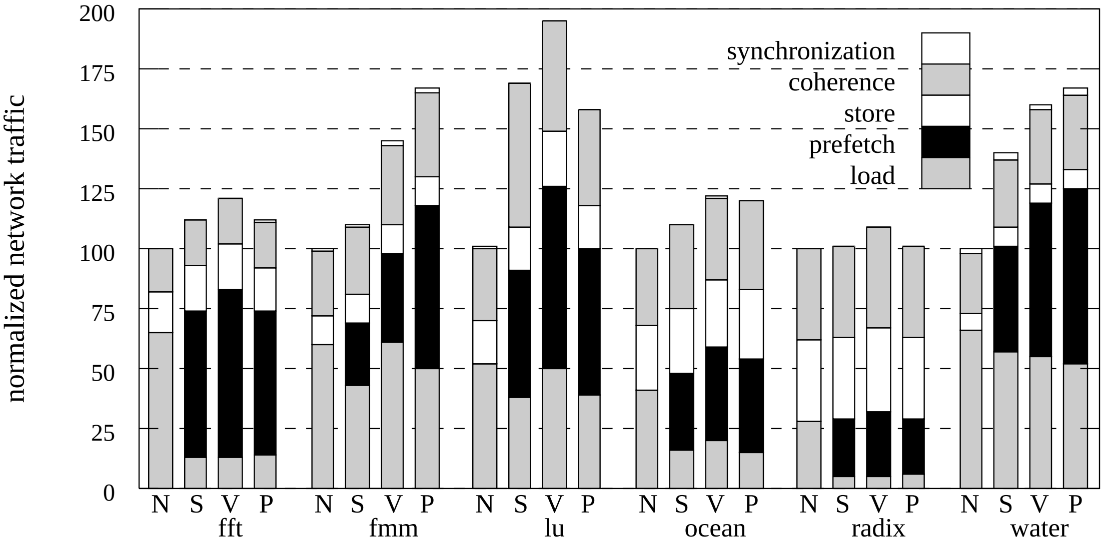

Figure 6: Network traffic with different prefetching schemes: All data are normalised to the result of no-prefetching. (`N`= no-prefetching, `S`= stride prefetching, `V`= virtual address DRT, `P`= physical address DRT)

5.3 Performance impact of the number of DRT entries

The previous subsections assumed a DRT with 128 entries. This subsection presents the impact of the number of DRT entries.

Figure 8 shows load stall times with different numbers of DRT entries. As expected, a smaller number of DRT entries reduce the effectiveness of prefetching. `Fmm, lu` and `water` cases are not affected much; the reason being that these benchmark program do not have regular strides as figure 4 shows. These programs gain little from stride prefetching. On the other hand, `fft, ocean` and `radix` cases need an adequate number of DRT entries. The results show that the number of DRT entries can be quite small to be effective. A DRT with 8 or 16 entries results in a performance quite close to a 128 entries for all benchmark programs. This is because the DRT only needs to be large enough to hold the active pages of the most frequently executed loop to be effective.

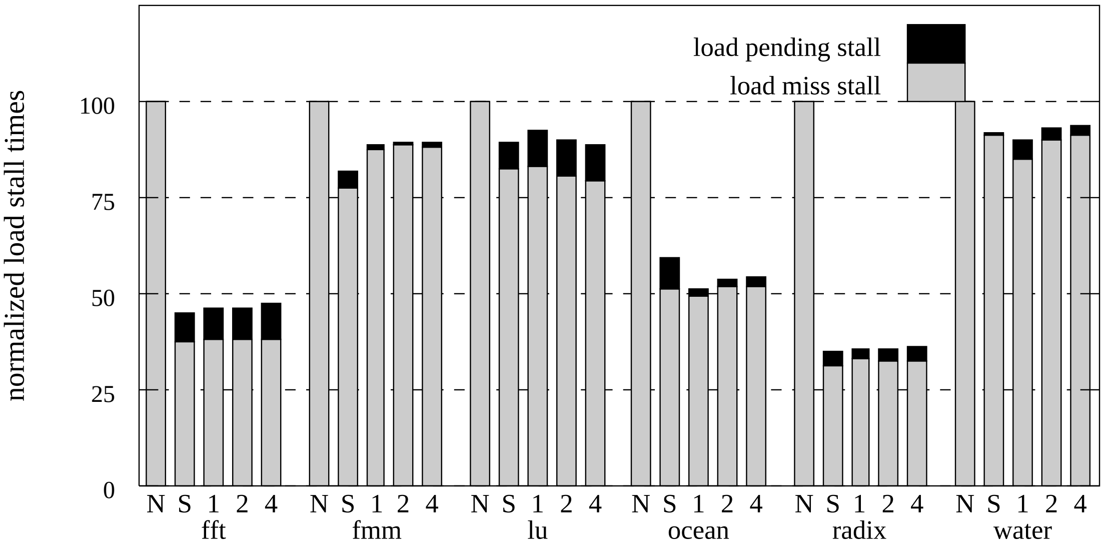

Figure 7: Load stall times with different page partitions: All data are normalised to the result of no-prefetching. (`N`= no-prefetching, `S`= stride prefetching, `1`= physical DRT with one partition, `2`= with two partition, `4`= with four partition)

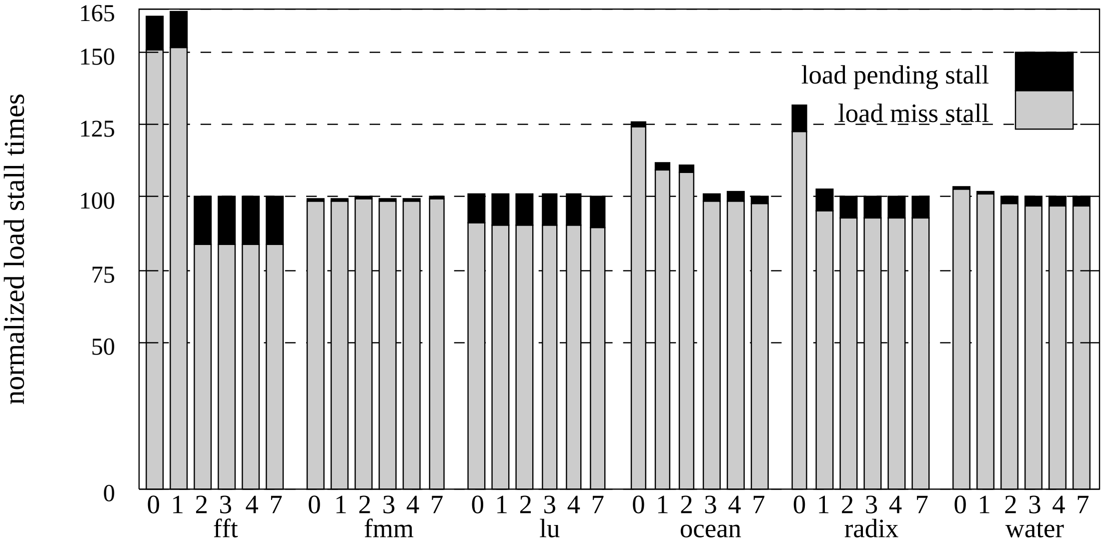

Figure 8: Load stall times with different number of DRT entries: All data are normalised to the result of 128 entries. (`0`= physical DRT with 1 entry, `1`= with 2 entries, `2`= with 4 entries, `3`= with 8 entries, `4`= with 16 entries, `7`= with 128 entries)

6 Related work

Many hardware data prefetching methods have been proposed. Only a few of them are for the secondary cache based on stride. In this section, we give a brief survey of secondary cache data prefetching using stride.

The stride directed prefetching for secondary cache was studied by Hagersten[8] in *right-on-time* (henceforth referred as ROT) prefetching. ROT uses the address of accesses missing in the primary cache instead of every operand address. The ROT consists of three main parts: stride detection, stream detection and prefetch. The stride detection part detects new strides. The address of each primary cache miss is compared with recent misses, all possible strides are calculated, and the occurrence of strides are recorded. If the occurrence frequency of a stride candidate passes a threshold, it is promoted to become the current predicted stride. The stream detection part maintains possible streams that are made when new strides are promoted by the stride detection part. When the contents of one of the streams and the primary cache miss address are the same, the prefetch part issues a prefetch using the stride of the stream. The hardware requirements, that include more than three hardware list tables and complex comparators, are relatively high. The results presented by Hagersten were incomplete because he studied his scheme under the unit stride without stride detection.

Dahlgren and Stenström evaluated stride and sequential prefetching in [4]. In their study, they compared the performance of two stride detection mechanisms. One was instruction address driven and the other was operand address driven. The latter was same as that proposed by Hagersten[8]. They pointed out that sequential prefetching does better or at least equally well as stride prefetching. However, stride prefetching has the potential to outperform sequential prefetching if network/memory bandwidth is limited, because stride prefetching can make fewer useless prefetches. They also showed that instruction address driven stride prefetching worked as well as, or better, than that of operand address driven.

Palacharla and Kessler studied a stride detection method for a stream buffer[16]. The stride detection scheme uses partitions of the physical address space and a history buffer. The partitions enable the separation of accesses in different streams and to detect strides from accessing in the same partition. They studied prefetching for stream buffer and uniprocessor system.

The ROT method, proposed in [8] and studied in [4], needs complex hardware that includes complex hardware list tables and comparators. In contrast, our method only needs a simple hardware table. Our experimental results show that a simple method, like the physical address data-range-table method, can achieve a performance improvement comparable to that of conventional stride prefetching, despite its much simpler hardware mechanisms. Palacharla and Kessler's method[16] is quite similar to ours. However, they did not think about software assisted method as our virtual address DRT nor did they think about hardware only approach as our physical address DRT, and their experiments were only about uniprocessor and stream buffer. On the other hand, we proposed both hardware and software approaches, and we did our experiments for cache and multiprocessor system.

7 Conclusion

Conventional high-performance computers usually use a two-level cache hierarchy that consists of primary on-chip cache and secondary off-chip cache. Cache hierarchy is very helpful to reduce the effective memory access time. However, processors must stall until the required data arrives if it is not found in the cache. To overcome this problem, prefetching techniques are used and they bring data to the cache before it is actually used. Therefore the key point of prefetching is deciding when and what to prefetch. There are two main hardware approaches. One is sequential prefetching that is based on spatial locality, and the other is stride prefetching based on stride calculation. From the viewpoint of prediction of which data will be referenced in the future, the latter approach is better. However, stride calculation needs access to the program counter. This makes it difficult for the secondary off-chip cache to use a stride prefetching method because the contents of the program counter are not visible outside a standard processor chip.

In this work, we considered a stride prefetching method for the secondary cache without accessing the processor's program counter. This method uses a data-range-table (DRT) and it enables the prefetching hardware to detect a memory access stream which falls into the same data range. We studied two variations of the DRT method. The first one uses virtual addresses and needs a system function that passes the data range information to the prefetching hardware. The second one uses physical address and need no software and system support.

We used program driven simulation of scientific applications from the SPLASH-2 suite in the context of shared-memory multiprocessors. Our evaluation results show that the proposed method can reduce load stall times by an amount comparable to those obtained by a conventional stride driven prefetching technique that needs access to instruction addresses, although our method needs much simpler hardware than others. A closer look at the benchmark programs used suggests that a DRT of eight to sixteen entries is sufficient for virtual address DRT. Simulation results show that DRT of sixteen entries is sufficient for physical address DRT. From the implementation standpoint, the DRT method is easy to realize. For example, it does not require any change of cache nor does it need access to the processor's internal resources.

Acknowledgement

We would like to thank *Manchester Computing* for providing us access to the IBM *Computer Intensive Facility* which consists of IBM RS6000 computers.
Ando Ki is supported by ETRI (Electronics and Telecommunications Research Institute, the Republic of Korea) and the British Council.

Appendix Proofs

A Detectable stride from the primary cache misses

If the primary cache *line size* is L and the access stream has *stride* S, the address difference (called *detectable stride* S') of two consecutive primary cache misses is some integer number between S and $S \times \lceil \frac{L}{S} \rceil$, inclusively. (i.e. $S \leq S' \leq S \times \lceil \frac{L}{S} \rceil$)

Proof: Without loss of generality, we can say the first cache line starts address 0. Therefore, the range of the first cache line is 0 to $L-1$, and that of the second cache line is L to $2L-1$.

Let assume the first primary cache miss occurs at the address δ:

$$0 \leq \delta < L \tag{1}$$

The following access addresses are $\delta + \gamma S$, where γ is an integer number greater than or equal to 1.

The address of the second primary cache miss must satisfy following equation:

$$L \leq \delta + \gamma S < L + S \tag{2}$$

The address difference of two consecutive primary cache misses is the value obtained by subtracting δ from $\delta + \gamma S$. Therefore, the range of detectable stride is

$$L - \delta \leq \gamma S < L + S - \delta \tag{3}$$

The maximum value of δ is $L-1$, and equation (3) is

$$1 \leq \gamma S < S + 1 \tag{4}$$

The possible integer γ, which satisfy the equation (4) is 1. This means the minimum detectable stride is S.

The minimum value of δ is 0, and the equation (3) is

$$L \leq \gamma S < L + S \tag{5}$$

The possible integer γ, which satisfy the equation (5) is $\lceil \frac{L}{S} \rceil$. This means the maximum detectable stride is $S \lceil \frac{L}{S} \rceil$.

Thus, that the detectable stride S' is $S \leq S' \leq S \times \lceil \frac{L}{S} \rceil$, is proved. ■

B Proof of $\mathbf{S \times \lceil \frac{L}{S} \rceil < L + S}$ when $\mathbf{0 < S < L}$

Proof: Let $L = nS + m$, where n is integer larger than or equal to 1 and $0 \leq m < S$.

For case 1, let assume $m = 0$, and therefore $L = nS$ and $\lceil \frac{L}{S} \rceil = n$.

$$\begin{aligned} (L+S) - S\left\lceil \frac{L}{S} \right\rceil &= (n+1)S - nS \\ &= S \\ &> 0 \end{aligned}$$

because $0 < S$. Therefore, case 1 is proved.

For case 2, let assume $0 < m < S$, and therefore $\lceil \frac{L}{S} \rceil = n + 1$.

$$\begin{aligned} (L+S) - S\left\lceil \frac{L}{S} \right\rceil &= (n+1)S + m - (n+1)S \\ &= m \\ &> 0 \end{aligned}$$

because $0 < m$. Therefore, case 2 is proved.

Thus, $S \lceil \frac{L}{S} \rceil < L + S$ is proved. ■

References

[1] Baer, J.-L., and Wang, W.-H. Architectural choices for multilevel cache hierarchies. In *International Conference on Parallel Processing* (Aug. 1987), pp. 258–261.

[2] Chen, T.-F. *Data Prefetching for High-Performance Processors.* Ph.D. thesis, University of Washington, 1993. Department of Computer Science and Engineering.

[3] Dahlgren, F., Dubois, M., and Stenström, P. Sequential hardware prefetching in shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems 6*, 7 (July 1995), 733–746.

[4] Dahlgren, F., and Stenström, P. Evaluation of hardware-based stride and sequential prefetching in shared-memory multiprocessors. In *International Conference on High-Performance Computer Architecture* (Jan. 1995), pp. 68–77.

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

[6] Fu, J. W. C., Patel, J. H., and Janssens, B. L. Stride directed prefetching in scalar processors. In *Proceedings of the 25th International Symposium on Microarchitecture* (1992), pp. 102–110.

[7] GHARACHORLOO, K., LENOSKI, D., LAUDON, J., GIBBONS, P., GUPTA, A., AND HENNESSY, J. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *17th ISCA* (June 1990), pp. 15–26.

[8] HAGERSTEN, E. *Toward Scalable Cache Only Memory Architectures.* PhD thesis, Royal Institute of Technology, Sweden, July 1993. Department of Telecommunication and Computer Systems.

[9] JOUPPI, N. P. Improving direct-mapped cache performance by the addition of a small full-associative cache and prefetch buffers. In *17th ISCA* (1990), pp. 364–373.

[10] KI, A. MUSIC Processor: Manchester university simplified instruction code processor. internal report, University of Manchester, 1995. Department of Computer Science.

[11] KI, A. Aspects of hardware-controlled data prefetching. internal report, University of Manchester, 1996. Department of Computer Science.

[12] KROFT, D. Lockup-free instruction fetch/prefetch cache organization. In *8th ISCA* (1981), pp. 81–87.

[13] LUSK, E., BOYLE, J., BNTLER, R., DISZ, T., GLICKFELD, B., OVERBEEK, R., PATTERSON, J., AND STEVENS, R. *Portable Programs for Parallel Processors.* Holt, Rinehar and Winston, Inc., 1987.

[14] MANCHESTER COMPUTING. *An Introduction to the IBM Compute Intensive Facility*, 1994. RSS101.

[15] MOWRY, T. C., AND GUPTA, A. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing 12* (1991), 87–106.

[16] PALACHARLA, S., AND KESSLER, R. E. Evaluating stream buffers as a secondary cache replacement. In *21st ISCA* (1994), pp. 24–33.

[17] SKLENAR, I. Prefetch unit for vector operations on scalar computers. *Computer Architecture News 20*, 4 (Sept. 1992), 31–37.

[18] SMITH, A. J. Sequential program prefetching in memory hierarchies. *IEEE Computer 11*, 2 (Dec. 1978), 7–21.

[19] WOO, S. C., OHARA, M., TORRIE, E., SINGH, J. P., AND GUPTA, A. The SPLASH-2 Programs: Characterization and methodological considerations. In *22nd ISCA* (1995), pp. 24–36.