# Hybrid Compiler/Hardware Prefetching for Multiprocessors Using Low-Overhead Cache Miss Traps

Jonas Skeppstedt

Dept. of Computer Engineering
Chalmers University of Technology
SE-412 96 Gothenburg, SWEDEN
jonas@acm.org

Michel Dubois

Dept. of Electrical Engineering-Systems
University of Southern California
Los Angeles, CA 90089-2562, USA
dubois@paris.usc.edu

## Abstract

*We propose and evaluate a new data prefetching technique for cache coherent multiprocessors. Prefetches are issued by a prefetch engine which is controlled by the compiler. Second-level cache misses generate cache miss traps, and start the prefetch engine in a trap handler generated by the compiler. The only instruction overhead in our approach is when a trap handler terminates after data arrives. We present the functionality of the prefetch engine and a compiler algorithm to control it. We also study emulation of the prefetch engine in software. Our techniques are evaluated on six parallel applications using a compiler which incorporates our algorithm and a simulated multiprocessor. The prefetch engines remove up to 67% of the memory access stall time at an instruction overhead less than 0.42%. The emulated prefetch engines remove in general less stall time at a higher instruction overhead.*

## 1 Introduction

Data prefetching is a promising approach to hide memory access latencies in cache coherent multiprocessors. Both compiler-based and hardware-based prefetch techniques have been studied which are effective for regular array accesses [5, 7, 8, 13, 14, 15]. In traditional compiler-based prefetching, the compiler tries to predict which memory accesses will suffer cache misses and it then inserts prefetch instructions for that data. Instruction overhead is a fundamental limitation of traditional compiler-based prefetching, and in [15] the importance of limiting the prefetching to accesses which are likely to experience cache misses was shown. Unfortunately, both the cache size and the input data set size (or loop iteration count) must be known at compile-time to predict cache misses accurately. In contrast, hardware-based prefetch techniques execute in parallel and do not cause any instruction overhead. Stride prefetching in hardware identifies the stride by comparing the sequence of addresses generated by each memory access instruction. However, the stride-detection phase requires quite complex hardware.

The problems with the previous techniques have motivated us to consider a hybrid prefetching technique which can take advantage of static compiler analysis to determine what to prefetch while still being able to issue prefetches with very little instruction overhead, and in particular, *no* instruction overhead cost at all when memory accesses hit in the cache.

In this paper we propose and evaluate a new prefetching technique, which is based on static compiler analysis, memory access traps, and a prefetch engine which issues prefetch requests. Our approach uses low-overhead cache miss traps as proposed in [10]. For certain memory access instructions, we let a miss in the second-level cache generate a *cache miss trap*. Each such memory access instruction has a corresponding trap handler which is generated by the compiler. The prefetch engine is started by trap handlers.

We also study whether one could emulate the prefetch engine in software by a loop in the trap handler which issues prefetches using ordinary prefetch instructions.

To evaluate our techniques we have implemented the prefetch engine in a detailed architectural simulator of a sequentially consistent cache-coherent multiprocessor and compiled six parallel scientific and engineering applications using an optimising compiler which incorporates our algorithm to generate cache miss trap handlers. We find that the prefetch engine removed up to 67% of the memory access stall time at an instruction overhead less than 0.42%. The emulated prefetch engine removes in general less stall time at a higher instruction overhead.

The rest of the paper is organised as follows. In Section 2 we give an overview of our prefetching technique. In Section 3 we present the functionality of the prefetch engine and in Section 4 we describe a compiler algorithm that generates the cache miss trap handlers to control prefetching. In Section 5 we present the experimental methodology, and we show the simulation results in Section 6. We discuss our results and relate them to work by others in Section 7. Finally, we conclude the paper in Section 8.

## 2 Prefetching Approach

The purpose of this section is to give the reader an idea of the types of memory access patterns which are handled by our prefetching approach.

The compiler marks certain memory instructions in a program to generate a low-overhead trap on a second-level cache miss. Such a memory access instruction is called a *faulting* memory access instruction, and for each faulting memory access instruction, there is a corresponding cache miss trap handler. Upon a cache miss trap, a prologue code

sequence saves a small number of the general purpose registers, locates the faulting instruction's trap handler using the saved program counter in a hash table, and jumps to the trap handler. After the trap handler has completed, an epilogue restores the saved registers and re-executes the faulted instruction. If the faulted instruction gets a second cache miss when it is re-executed, it does not generate a new trap.

The compiler controls a prefetch engine by specifiying the following prefetch parameters: *Count* is the number of prefetches to issue using direct addressing, *Stride* is the distance in bytes between blocks that are prefetched, *Indirect* is the number of prefetches to issue using indirect addressing, *Cache state* specifies whether a block is expected to be read only or also modified, and finally, *Address* is the byte address of the first prefetch. So, the task of the compiler is to extract these parameters for each stride access in a loop.

To illustrate the operation of a prefetch engine, we will use C code fragments and show how the engine should be controlled. In the examples below we assume that the cache block size is *B* bytes.

```
for (i = 0; i < n; i = i + D)
        x = x + a[K*i];
```

Figure 1: Example code which reads an array.

In the code in Figure 1, a number of elements of an array `a` are read. A trap handler is associated with the instruction that loads `a[K*i]`, and will be invoked when the load instruction generates a cache miss trap. The information that the compiler extracts from the code in Figure 1 is the stride of `a[K*i]`, the number of blocks to prefetch, and the starting address. While the stride is constant and is required to be known at compile-time, the number of prefetches to issue is computed in the trap handler. The number of prefetches depends on the value of the loop index `i` when a miss occurs. The task for the compiler is to generate code for the trap handler that computes this by comparing `i` and `n`. The index `i` is incremented by `D` in each loop iteration, and the number of iterations remaining in the loop is $(n - i)/D$. An address-expression with a stride must contain a variable that is incremented by a constant in each loop iteration. In Figure 1, this variable is `i`. Assuming that the size of an array-element is $E$, the stride *S* of the instruction loading `a[K*i]` then becomes $K \times D \times E$. If the stride *S* is equal to or greater than the cache block size *B*, then $N = (n - i)/D$ blocks will be accessed. On the other hand, if *S* is less than *B*, the number of cache blocks that will be accessed becomes $N = \lceil (n - i) \times S/(D \times B) \rceil$ (assuming $i$ refers to the beginning of a cache block). We should prefetch one block less than *N* since the missing load instruction requests the first block itself.

In Figure 2 we give an example of indirect prefetching. `a` is an array of pointers to integers and each loop iteration dereferences one pointer. In this case the prefetch engine is set up to do indirect prefetching. When the engine has prefetched one cache block of the array `a` using direct addressing, it will prefetch the pointed-to blocks as well. While we not describe the details, indirect addressing can also be used to prefetch blocks pointed-to by record fields.

```
int     *a[];

for (i = 0; i < n; i++)
        x = x + *a[i];
```

Figure 2: Here indirect addressing is used.

To emulate prefetch engines in trap handlers, a trap handler issues prefetches using ordinary prefetch instructions in a loop. In order to reduce the risk of hot-spots in the memory system, the number of prefetches is limited to four. Indirect addressing prefetch is not emulated by trap handlers. Apart from this, the compiler analysis to generate trap handlers is identical regardless of whether the trap handler will start a prefetch engine or it will emulate a prefetch engine.

# 3 Prefetch Engine Functionality

This section describes the state variables and the operation of the prefetch engine that we have evaluated. A processor has multiple prefetch engines, and the exact number is implementation-defined. Before the compiler can specify prefetch parameters, one prefetch engine must first be chosen. This is done with a *reset* command which selects one prefetch engine. Subsequent parameters up to and including the start address will implicitly refer to the most recently selected engine.

## 3.1 Prefetch buffer

The purpose of the prefetch buffer is to control the prefetch issue pace. Each prefetch engine has a prefetch buffer with four entries and a prefetch cannot be issued unless there is a free entry. The motivation for using four entries is that, according to Mowry's measurements [13], it does not pay to allow for more than four outstanding prefetches. The state for each entry is shown in Table 1. A prefetch request allocates an entry and records the block address. An entry is deallocated when two conditions are satisfied: data has arrived and it has been requested by the processor. A replacement or an invalidation also deallocates an entry. A reset command deallocates entries whose data has arrived but that has not yet been accessed (otherwise, a useless prefetch would occupy an entry until the entry is deallocated by a replacement or an invalidation). The *Accessed*-flag is set when an access refers to a block with a pending prefetch, and is reset when an entry is allocated.

| Name | Description |
|---|---|
| Free | True if entry is not in use |
| Accessed | True if block has been accessed |
| Block | Prefetched block number |

Table 1: Prefetch buffer entry.

## 3.2 Prefetch engine state

In Table 2 we show which state variables a prefetch engine uses. There are three groups of state variables. The first group, *Operating* and *Stride*, is used both for direct

and indirect address prefetching. The next group, *Address*, *Count*, and *DState*, is used only for direct addressing, and the last group *Pointer*, *Indirect*, and *IState*, is used only for indirect addressing. We will now describe the purpose of each variable.

To start with the first group, *Operating* is true when the engine has valid prefetch parameters. *Stride* is the amount that both *Address* and *Pointer* will be incremented for the next prefetch using direct addressing. In the next group, *Address* is the byte address of the next block to prefetch using direct addressing, and *Count* is the number of remaining prefetches to issue. *DState* controls whether shared or exclusive mode prefetches should be issued for direct addressing. In the last group, *Pointer* is the byte address of a pointer in the program. To issue a request using indirect addressing, the data that contains the address must (of course) be present in the cache first. *Indirect* is the number of blocks to prefetch starting at the memory (cache) contents at *Pointer*, and *IState* specifies whether shared or exclusive mode prefetches should be used. *Indirect* specifies the number of sequential blocks to prefetch using indirect addressing. We will now describe how direct and indirect prefetching is carried out.

| Name | Description |
|---|---|
| Operating | True if parameters are valid |
| Stride | Byte stride |
| Address | Next address to prefetch |
| Count | Remaining prefetches to issue |
| DState | Direct addressing cache state |
| Pointer | Next address to prefetch indirectly |
| Indirect | Number of indirect prefetches |
| IState | Indirect addressing cache state |

Table 2: State variables in each prefetch engine.

### 3.3 Prefetch engine operation

A reset selects an engine, deallocates the engine's prefetch buffer entries as discussed in Section 3.2, clears all state variables of that engine, and initialises the following default values: *Stride* is set to the cache block size, the addressing mode is set to direct addressing by setting *Indirect* to zero, *Count* is set to a maximum value (we use the number of cache blocks in a page), and both direct and indirect addressing prefetch are set to use shared state. After a reset, the default values may be overridden by giving new values. Finally, when the *Address* parameter is specified, the *Address* and *Pointer* state variables are set to this address and *Operating* is set to true. An operating prefetch engine issues prefetches using direct addressing until its *Count* becomes zero or it is selected by another reset command. We will first describe the operation of direct addressing prefetch and then indirect addressing prefetch. A new prefetch request can be issued by an engine when a free entry in the prefetch buffer is available.

The *Address* state variable contains the byte address of the next block to prefetch. This block is looked-up in the second-level cache and if it is not present, the block is requested and a prefetch buffer entry is allocated. If the block was prefetched or present, the *Address* variable is incremented by *Stride* and *Count* is decremented by one. *Address* and *Count* do not change if the block was absent but there was no available buffer entry. If *Address* crosses a physical page, *Count* is set to zero. This will generate a new trap at a cache miss in the new page and prefetching will start again.

The state variable *Pointer* contains the byte address of a pointer in the program, and is initialised to the same value as *Address*. If the block of the pointer is not present in the cache, indirect addressing prefetching is paused until that block arrives, eg as a result of direct addressing prefetch. When the block is present, the cache content at the memory address *Pointer* is read and is treated as a virtual address *VA*. *VA* is translated to a physical address and then a prefetch is issued when a free entry becomes available. Note that indirect prefetch requires interaction with the virtual memory system.

To limit the prefetching activity, an engine has only four prefetch buffer entries. Both direct and indirect addressing compete for these four entries and we give indirect addressing higher priority to be granted an entry.

## 4 Compiler Algorithm

This section gives an overview of a compiler algorithm for automatically generating cache miss trap handlers for prefetching. The analysis performed by the compiler is based on natural loop analysis, induction-variable analysis, and a dataflow analysis similar to live-variables analysis [1]. In general, our algorithm can detect stride accesses in loops if an induction-variable is part of an access's address-expression.

#### Direct addressing engine prefetch

Assume a memory access *A* belongs to a natural loop *L*. If *A* will generate addresses that differ by a constant *S* (known at compile-time) in subsequent iterations of *L*, then *A* is a *stride access*. The constant *S* denotes the *stride*. A trap handler is generated for every stride access. If *L* has a loop-termination condition of the form $i < n$, the remaining number of loop iterations can be computed and from that the number of prefetches to issue. Assume $i$ is incremented by $D$ in each iteration. As we discussed in Section 2, if the stride *S* is equal to or greater than the cache block size *B*, the number of direct addressing prefetches, $N$, is set to $(n - i)/D - 1$. Otherwise, if *S* is less than *B*, $N$ is set to $(n - i) \times S/(B \times D) - 1$. Here $i$ and possibly $n$ are variables, and $D$, $B$, and $S$ are compile-time constants. The number of prefetches is not computed using multiply and divide instructions, which can be too time-consuming, instead we approximate the number using shifts.

If the number of prefetches to issue cannot be computed at runtime, a default number is used instead.

#### Indirect addressing engine prefetch

For indirect addressing prefetch, the compiler must analyse the use of data read by a load instruction. If the data read by one load instruction $A_1$ is used as a base address by another memory access instruction $A_2$, then $A_1$ is said to be a *parent* of $A_2$, which is said to be a *child*. When the parent is a stride access, then indirect engine prefetching is started in the trap handler of the parent. The engine parameter *Indirect* is

determined by considering which blocks are accessed using the base pointer loaded by the parent instruction.

# 5 Experimental Methodology

To evaluate our prefetch technique, we have incorporated the prefetch algorithm in a compiler. We have then compiled and run six parallel applications on a simulated cache-coherent NUMA multiprocessor. First we present the compiler and the benchmarks we have used. Then we present the multiprocessor architectures we have simulated to evaluate effects on execution time and traffic.

## 5.1 Compiler and benchmark programs

We have incorporated the compiler algorithms in an optimising C compiler [17] which compiles parallel applications using the ANL macros [2] and generates code for shared-memory multiprocessors based on SPARC processors. We have used a set of six applications developed at Stanford University (Water, Cholesky, LU, MP3D, Barnes-Hut, and PTHOR), of which all but LU are part of the SPLASH-1 suite [16]. We used the data set sizes that are shown in Table 3.

Table 3: Benchmark Programs, Data Set Sizes used.

| Benchmark | Data Sets |
|---|---|
| Water | 244 molecules, 3 time steps |
| Cholesky | matrix bcsstk14 |
| MP3D | 50,000 particles, 5 time steps |
| LU | 200 x 200 matrix |
| Barnes-Hut | 128 bodies |
| PTHOR | RISC circuit |

## 5.2 Metrics of Detection Efficiency

To understand how close to the optimum the prefetch efficiency of the compiler algorithm is, we have measured the number of second-level cache misses in an execution that were removed because the data was prefetched. Unless the processor was stalled waiting for a prefetched block $B$, when $B$ is loaded into the second-level cache (*SLC*), a *PF*-flag is set in the simulator's cache block. When the block is accessed, invalidated, or replaced, the flag is reset. Let *M* be the number of *SLC* misses which request data from memory (this excludes misses to blocks that have a pending prefetch and also re-executed faulted instructions). Let *H* be the number of *SLC* accesses where the *PF*-flag is true and *P* be the number of *SLC* accesses to blocks which have a pending prefetch. Consequently, *H* is the number of memory accesses whose latency was hidden, and *P* is the number of memory access whose latency was partly hidden, and they count the number of misses which are covered by prefetches. We define the *coverage* to be $C = (H + P)/(H + P + M)$.

A prefetch request sent to memory which does not cover a cache miss is useless. We define the *degree of bad prefetches* to be the number of useless prefetch requests divided by the number of prefetch requests sent to memory.

The measurements coverage, useless, execution-time, and traffic have all been carried out by executing the compiled applications on a detailed architectural simulator which we will describe next.

## 5.3 Simulated architectures

We have developed two detailed architectural simulation models: first a basic write-invalidate protocol which constitutes the baseline architecture and second the baseline extended with load and store instructions that can generate a cache-miss trap and with prefetch engines. These models are described in detail below. The simulation platform consists of a functional simulator of SPARC processors which generate memory references to an attached memory system architectural simulator with a detailed timing model [3]. Since the executing processors are delayed according to the latencies encountered by each memory reference, the same interleaving of memory references will be encountered as in the target architecture.

### Baseline architecture

The overall organisation of the baseline architecture is shown in Figure 3. It consists of 16 processing nodes. Apart from the local portion of the shared memory, each processing node also contains a two-level cache hierarchy whose organisation is shown in Figure 4. It consists of a write-through, direct-mapped first-level cache (denoted *FLC*) with an associated first-level write buffer denoted *FLWB*. In the baseline, the *FLWB* buffers requests to a copy-back, second-level cache (*SLC*). Since the processor is stalled on loads that miss in the *FLC* and on stores, the *FLWB* is not needed in the baseline architecture. As we will see below, it is used to buffer requests that do not need to stall the processor, namely, requests related to prefetching.

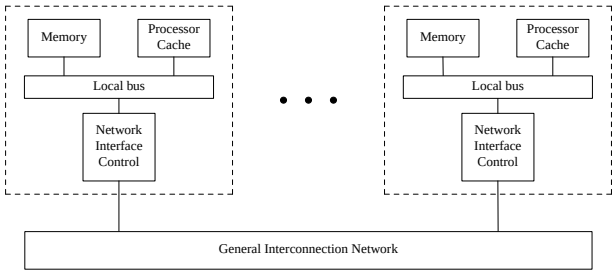


Figure 3: Organisation of a cache coherent multiprocessor.

System-level cache coherence between the second-level caches is maintained by a Censier and Feautrier write-invalidate protocol which associates a bit vector with each memory block [4]. Virtual pages are 4KB and are mapped to physical memory modules using a round-robin policy that interprets the four least significant bits of the virtual page number as the node identity. The node in which a certain page is mapped is called the *home* of all blocks in that page.

Loads that miss in the *FLC* and the *SLC* cause a miss request to be sent to home. If the copy is present at home, and if home is the local node, the miss is serviced locally. Otherwise, two or four node-to-node traversals are required to fill the cache.

Stores are written through the *FLC*. If the *SLC* copy is exclusive, the store can be carried out locally. Otherwise, ownership has to be acquired. The coherence protocol in both the baseline and the extended architecture we evaluate, implements *sequential consistency* by stalling the processor

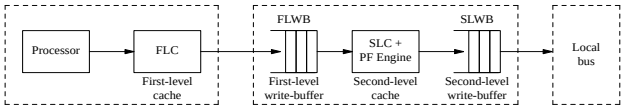


Figure 4: The two-level cache hierarchy in each processing node. The prefetch engine is part of the SLC.

until ownership is granted. Depending on the location of home and whether another node has an exclusive copy, ownership acquisition may encounter zero, two, or four node-to-node traversals.

In Figure 4, a write buffer is also associated with the *SLC*, denoted *SLWB*. Since the processor is stalled on every global store, this buffer is not needed in the baseline architecture.

#### Faulting memory access instructions

We evaluate the effectiveness of our compiler algorithms by replacing marked memory accesses by special instructions denoted *faulting memory access instructions*. Unlike ordinary memory access instructions, they generate a hardware cache miss trap on a second-level cache miss.

The actions taken by the cache hierarchy when an ordinary load or store instruction executes are identical to those of the baseline. The actions taken by the cache hierarchy when a faulting memory access instruction executes are as follows. If the block is present in the *FLC*, the behaviour is identical with that of the baseline. If the block is not present in the *FLC*, however, the processor has to stall, and the memory request is buffered in the *FLWB*. If the *SLC* has a copy, the behaviour again is identical with that of the baseline. Conversely, if the block is not present in the *SLC* and there is no pending request in the *SLWB* (or in a prefetch buffer), a *cache miss trap* is generated.

The request of the cache miss which generated the trap is recorded in the *SLWB*, the request is sent to home, and the processor continues execution in a trap handler prologue. To support faulting memory access instructions, the *SLWB* should have at least two entries; one entry for the faulted access and one entry if an access in the trap handler would experience another second-level cache miss. A cache miss trap is a low-overhead trap [9], and a trap does not need service from the operating system. During execution of a trap handler, new memory access traps are disabled, and when an instruction is re-executed, it behaves exactly as in the baseline architecture.

#### Support for prefetch engines

An *SLC* is extended with four prefetch engines where each engine has a prefetch buffer with four entries. To issue a prefetch by an engine, the request must first be allocated a prefetch buffer entry. An engine request cannot allocate an entry whose block has not been accessed; the purpose of this is to control the prefetch issue pace. Indirect prefetch requires a TLB access. In this study, we do not model any cost for doing this translation. Instructions to control the prefetch engine and ordinary prefetch instructions do not stall the processor; rather, they are buffered in the *FLWB*.

#### Support for emulated prefetch engines

The hardware support required to emulate prefetch engines in trap handlers is as follows: faulting memory access instructions, a lockup-free *SLC*, and ordinary prefetch instructions included in modern instruction set architectures. To make the *SLC* lockup-free, pending prefetch requests are buffered in the *SLWB*. A prefetch request, either from a prefetch engine or an ordinary prefetch instruction, first checks that a block is not in the *SLC* already or has a pending request.

#### Architectural parameters

In our simulations we assume that the *FLWB* and the *SLWB* contain 8 and 16 entries, respectively. The architectural parameters we assume for all three architecture variations are as follows. Each node contains a SPARC processor clocked at 200MHz (1 pclock = 5ns). We model a 4KB *FLC* and a 64KB *SLC*, both direct mapped and with a block size of 16 bytes. *FLC*, *SLC*, and local memory access times are 1, 6, and 30 pclocks, respectively. The nodes are interconnected with a network with a fixed node-to-node latency of 54 pclocks. Each control message is 5 bytes, and each data message is 21 bytes. Only shared references in the applications' parallel section are modelled with these parameters. Other memory accesses are assumed to hit in the *FLC*.

## 6 Simulation Results

In this section we first show the detection efficiences and then as a case study show the effects on the execution-times.

### 6.1 Detection Efficiency

The diagrams of Figure 5 show the coverages (top) and degrees of bad prefetches (bottom) for the applications we have studied. For each application we show five bars that from left to right correspond to the following prefetch techniques: prefetch engine using direct addressing only is *DH*, *DH* extended with indirect addressing prefetch is *IH*, *IH* extended with exclusive mode prefetching is *EH*, emulated prefetch engine using direct addressing is *DS*, and finally *DS* extended with exclusive mode prefetching is *ES*. We will also collectively refer to *DH*, *IH*, and *EH* as *HW*, and *DS* and *ES* as *SW*.

For three of the applications (Water, Cholesky, and LU), most cache misses are to arrays accessed with direct addressing. MP3D is an application with both regular array accesses and indirect accesses. Finally, the last two applications (Barnes-Hut and PTHOR) have many pointer dereferences. The compiler used indirect addressing prefetch only for three of the applications, namely, MP3D, Barnes-Hut, and PTHOR. So, for Water, Cholesky, and LU, the simulation results for *DH* and *IH* are identical.

We expect that the detection efficiency for *HW* should be somewhat better than for *SW* since *SW* is limited to prefetch at most four cache blocks per cache miss. An upper bound of the coverage for *SW* is therefore 80%, which is achieved when one miss causes four useful prefetches.

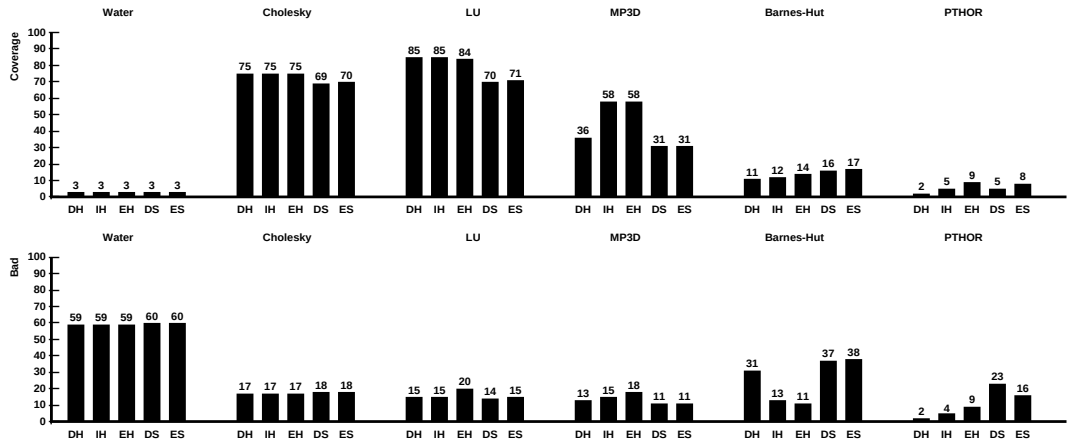


Figure 5: Coverage and degree of bad prefetches in percent.

To start with Water, the number of prefetches issued is low for each technique[1]. Of the few issued prefetches less than half were useful: each technique covered only 3% of the misses. We found two situations that limited the coverage. The first is in the procedure `INTERF` where one variable `comp` is used to index array elements in a loop. Although the value of `comp` normally is incremented by one for each iteration, in some cases its value is set modulo another variable, which prevents our compiler from using a prefetch engine for these accesses. The second situation which our compiler currently cannot handle is when the surrounding loop is in one source file and the array accesses are in another. With interprocedural analysis at the file level this situation could be handled. A significant fraction of the prefetched data was invalidated resulting in high degrees of bad of 59% for *HW*, and 60% for *SW*.

Continuing with Cholesky, all techniques use prefetching extensively and are quite successful. *HW* covers 75%, *DS* covers 69%, and *ES* covers 70% of the misses. The degrees of bad are 17% for *HW* and 18% for *SW*. These useless prefetches were due to prefetched data that was either replaced or invalidated.

For LU, prefetching is also used extensively. *HW* has a coverage which exceeds the upper bound that *SW* can reach. *DH* covers 85% and *EH* covers 84%. We analysed the reason why *HW* did not reach an even higher coverage and found it is mainly due to a limitation in our coherence protocol. When one processor has produced a column, then all processors waiting for this column will prefetch the cache blocks of that column. However, our coherence protocol permits only one prefetch request to wait for home to become clean. The other nodes receive a negative acknowledgement (nack) of their prefetch request. This limitation can be removed by a more sophisticated coherence protocol. These nacks constitute the majority of the useless prefetch requests both for *HW* and for *SW*. The remaining misses in LU were mostly to the synchronisation structure `Global->done`.

There are two data structures in MP3D, one array of particle records and one array of cell records. Each particle has a pointer to a cell. The cells are migratory objects and most misses are to the cells. With *DH* and *SW*, only the particles are prefetched, and therefore their coverages become only 36% and 31%, respectively. With *IH*, an engine prefetches the pointed-to cells as well, and a coverage of 58% is reached. The misses that remain are mostly due to prefetched cells that were invalidated before they were accessed, and to cells when a particle moves from one cell into another. As expected, *SW* has a lower degree of bad prefetches than *HW* since *SW* only prefetches particles (which are seldom invalidated).

Barnes-Hut is an application whose main data structure is recursive and is operated on by recursive procedure calls, which limit the prefetching. For *DH* and *SW*, typically only the array of subnode pointers are prefetched — but not the subnodes themselves. *IH* does prefetch subnodes using indirect addressing in the recursive procedure `walksub()` and reaches a marginally higher coverage of 12%. However, many misses remain that could not be handled by our algorithm.

PTHOR is also an application whose main data structure is recursive and is a graph of circuit elements. There are more prefetches issued for exclusive mode prefetching, which indicates that *EH* and *ES* create new misses. For this application *DH* and *IH* only cover 5% of the misses. The degree of bad prefetches for *ES* is 16%.

In summary, we find that the prefetch engine reaches high coverages for codes with regular array accesses, namely LU and Cholesky, and that the emulated engines reach coverages which are quite close to their upper bounds of 80%. The degrees of bad are around 20%. We also see that indirect engine prefetch could contribute significantly to the coverage for one application with pointer dereferences (MP3D), but many misses due to pointer dereferences that could not be handled by our algorithm remain. One other reason for not having an even higher coverage was a limitation in the coherence protocol we used, namely, that multiple prefetch requests for a block were allowed while home was waiting for becoming clean. Another limitation was related to the scheduling of prefetches. Some blocks were invalidated before they were used. We will next consider the effects on the execution time and traffic of our prefetch technique.

[1] The total number of prefetches cannot be derived from coverage and degree of bad but we measured that separately.

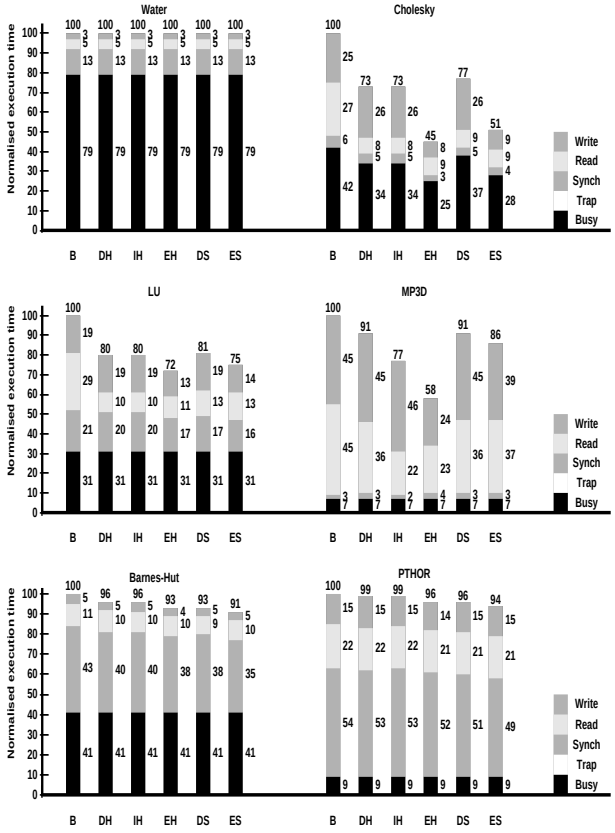


Figure 6: Normalised execution times.

### 6.2 Effects on Execution Time

In this section we present the execution times for the applications in Figure 6. For each application we show six bars where *B* is the baseline in addition to *DH*, *IH*, *EH*, *DS*, and *ES*, which are the same as in Section 6.1. We will again collectively refer to *DH*, *IH*, and *EH* as *HW*, and *DS* and *ES* as *SW*. The normalised execution times for each application is broken down into the following components from bottom to top: the busy time, the trap handler time, synchronisation and buffer stall, the read stall, and the write stall. We define the trap handler time as the execution time a processor is executing in a trap handler *after* that the data has arrived.

To start with Water, no prefetch technique has any effect on the the execution-time. This was due to the compiler could detect very few stride accesses, as discussed in Section 6.1.

Continuing with Cholesky, we see that the memory access stall accounts for more than half of the execution time in *B*. *DH* reduces the read stall time from 27% down to 8%. The synchronisation stall time is also reduced from 6% to 5%. *EH* reduces the write stall time from 25% to 8%. However, *E* has a somewhat longer read stall time than *DH*, 9% versus 8%. Although the execution time has dropped to 45% for *EH*, part of this is due to the application's scheduling which in this case reduces the busy time significantly.

For LU, *DH* and *DI* reduce the read stall time from 29% to 10%, and *EH* reduces it to 11%. *SW* does not reduce the read stall time to the same extent as *HW*, but still cuts it to less than half. *EH* and *ES* also reduce the write stall time from 19% down to 13% and 14%, respectively.

For MP3D, *DH* and *SW* are not expected to reduce the read stall time significantly because indirect addressing prefetch is required. *IH* on the other hand reduces it from 45% to 22%. *EH* also reduces the write stall time from 45% to 24%.

Finally, for Barnes-Hut and PTHOR, we see that both the read stall and the synchronisation stall are reduced slightly by each technique, and the write stall time is also reduced marginally for *EH*.

Instruction overhead is introduced when a trap handler completes after data arrives. The instruction overhead is defined to be the trap time divided by the busy time. For none of the applications the overhead exceeds 0.42%, and this is why the trap times don't appear in the diagrams. We also measured the amount of generated traffic and found that it is not affected much by our prefetch techinques. By removing separate ownership requests, for Cholesky, LU, MP3D, and Barnes-Hut, *EH* could reduce the traffic between 1% and 4%, while the worst increase for any technique and application was an increase by 10% for *ES* for Barnes-Hut. In summary, we see that data prefetching using prefetch engines —either implemented in hardware or emulated in software— are successful at reducing both the read and write stall time at very little instruction overhead and increased traffic.

## 7 Related Work

Compiler-based data prefetching has been studied extensively by Mowry in [13]. As mentioned in Section 1, one goal of this paper is to overcome the problem of instruction overhead present in his approach. The instruction overhead limits his approach to source codes where the compiler can predict that misses will occur. In our approach on the other hand, there is instruction overhead only in the trap handlers, but as we saw in Section 6, this overhead is very small, and more importantly is only present when misses occur. In our approach, there is no need to restrict the compiler's use of data prefetching.

In recent work, Luk and Mowry [12] have evaluated a compiler algorithm to prefetch recursive data structures. We expect that the prefetch engine approach presented in this paper will not be useful at prefetching recursive data structures, because of the difficulty at generating addresses to prefetch without actually traversing a recursive data structure, which our prefetch engines are not intended to do.

To reduce the complexity of pure hardware-based stride prefetchers is another motivation of this work. The stride-prefetcher proposed by Chen and Baer [5] includes complex hardware to analyse the string of accesses to detect strides, and the hardware includes a reference prediction table [5]. Our simulations indicate that this complexity is not necessary. Chen [6] has proposed an on-chip prefetch engine which works with the first-level cache and is programmed by a compiler before a loop is entered (although the compiler's task was done by hand in [6]). A difference between that work and ours is that our technique suffers no instruction overhead when there are no misses. However, Chen's prefetch engines can, of course, also exploit

low-overhead cache miss traps proposed in [10] and the compiler-generated trap-handlers proposed in this paper.

## 8 Conclusion

The contributions of this paper are the design and evaluation of a new approach to do data prefetching in multiprocessors. The components of our approach are a prefetch engine that issues prefetch requests, memory access instructions that trap on a second-level cache miss, and a compiler algorithm that automatically generates trap handlers. The prefetch engine is initialised by trap handlers with the number of blocks to prefetch, the access stride, and an address to start prefetching. Once started, the prefetch engine executes autonomously and creates no instruction overhead. We also evaluated the possibility of emulating the prefetch engine in a software loop in the trap handler. To evaluate our designs, we have implemented the prefetch engine in a detailed multiprocessor simulator, incorporated the compiler algorithm in an optimising compiler, and compiled and run six parallel applications. We find that the memory access stall time could be reduced by up to 67%, at an instruction overhead of less than 0.42% and very little additional memory traffic.

Although we have evaluated the prefetch engine in the context of a cache coherent NUMA, we expect that the prefetch engine can have a potential to reduce memory access latencies also in uniprocessors. We also expect that distributed virtual shared memory systems [11] could take advantage of our stride prefetching because each miss takes a large number of cycles. In this case, it might be good to "batch" the prefetches, so that they cause one single interruption on their way back.

To conclude, we have shown that with proper hardware support, it is possible to exploit an optimising compiler's static analysis in order to do accurate data prefetching at very little instruction overhead. In addition, we find that the emulated prefetch engine is competitive with prefetch engines while not requiring any hardware support beyond cache miss traps and prefetch instructions.

## Acknowledgements

This research has been supported by a grant from the Swedish Research Council on Engineering Science (TFR) under the contract number 94-315.

## References

[1] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., 1986.

[2] J. Boyle, R. Butler, T. Diaz, B. Glickfield, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors.* Holt, Rinehart, and Winston, New York, 1987.

[3] Mats Brorsson, Fredrik Dahlgren, Håkan Nilsson, and Per Stenström. The CacheMire Test Bench - A flexible and effective approach for simulation of multiprocessors. In *Proceedings of the 26th IEEE Annual Simulation Symposium*, pages 41–49. IEEE, New York, March 1993.

[4] Lucien Censier and Paul Feautrier. A new solution to coherence problems in multicache systems. *IEEE Trans. Comput.*, 27(12):1112–1118, 1978.

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

[6] Tien-Fu Chen. An effective programmable prefetch engine for on-chip caches. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 237–242, 1995.

[7] 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, 1995.

[8] Eric Hagersten. *Toward Scalable Cache Only Memory Architectures*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden., October 1992.

[9] Mark Horowitz, Margaret Martonosi, Todd Mowry, and Mike Smith. Informing Loads: Enabling Software to Observe and React to Memory Behavior. CSL-TR-95-673, Computer Systems Laboratory, Stanford Univ., July 1995.

[10] Mark Horowitz, Margaret Martonosi, Todd Mowry, and Mike Smith. Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 260–270. ACM, New York, 1996.

[11] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321-359, November 1989.

[12] Chi-Keung Luk and Todd C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233. ACM, New York, 1996.

[13] Todd Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford Univ., Computer Systems Laboratory, Stanford, Calif., March 1994.

[14] Todd Mowry and Anoop Gupta. Tolerating latency through software-controlled prefetching in scalable shared-memory multiprocessors. *J. Parallel Distrib. Comput.*, 2(4):87–106, 1991.

[15] Todd Mowry, Monica Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73. ACM, New York, 1992.

[16] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Comput. Arch. News*, 20(1):5–44, 1992.

[17] Jonas Skeppstedt. The design and implementation of an optimizing ANSI C compiler for SPARC. Technical report, Dep. of Computer Science, Lund Univ., Lund, Sweden, April 1990.