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

Jonas Skeppstedt and Michel Dubois

CENG Technical Report 97-04

Department of Electrical Engineering-Systems
University of Southern California
Los Angeles, California 90089-2562
(213)740-4475

February 1997

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

Jonas Skeppstedt and Michel Dubois

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

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

Abstract

In this paper we propose and evaluate a new data-prefetching technique for cache coherent multiprocessors. Prefetches are issued by a new functional unit called a *prefetch engine* which is controlled by the compiler. We let second-level cache misses generate cache miss traps, and start the prefetch engine in a trap handler. The trap handler is fast (40-50 cycles) and does not normally delay the program beyond the memory latency of the miss. Once started, the prefetch engine executes on its own and causes no instruction overhead. The only instruction overhead in our approach is when a trap handler completes after data arrives. The advantages of this technique are (1) it exploits static compiler analysis to determine what to prefetch which is hard to do in hardware, (2) it uses prefetching with very little instruction overhead, which is a limitation for traditional software-controlled prefetching, and (3) it is accurate in the sense that it generates very little useless traffic while maintaining a high prefetching coverage. We also study whether one could emulate the prefetch engine in software, which would not require any additional hardware beyond support for generating cache miss traps and ordinary prefetch instructions.

In this paper we present the functionality of the prefetch engine and a compiler algorithm to control it. We evaluate our technique on six parallel scientific and engineering applications using an optimising compiler with our algorithm and a simulated multiprocessor. We find that the prefetch engine removes 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.

Keywords: Data prefetching, memory access traps, cache coherent multiprocessors, compiler algorithms, performance evaluation.

Corresponding author: Jonas Skeppstedt

1 Introduction

Coherent data caches are key to achieving good performance in shared-memory multiprocessors. The majority of memory access instructions can be serviced by the data cache of a processing node, and this reduces the average memory access latency seen by the processor. Despite this, memory accesses often account for a significant fraction of the total execution time. This is due to cache misses that must be serviced by remote memory. One technique to hide memory access latency is to fetch data closer to the processor before it is actually needed, that is, data prefetching. A successful prefetching technique must be able to predict which data will be accessed in the near future. This is in general difficult [9], but for regular array accesses, both compiler-based and hardware-based prefetching techniques have been able to predict future accesses quite accurately [5, 7, 8, 13, 14, 15].

In traditional compiler-based prefetching, prefetch instructions are inserted into the code. While static compiler analysis can deal with more complex memory access patterns than can a pure hardware-based approach, instruction overhead is a fundamental limitation of traditional compiler-based prefetching. The instruction overhead arises partly due to the execution of prefetch instructions and partly due to address calculations and increased register pressure. In [15] the importance of limiting the prefetching to accesses which are likely to experience cache misses was shown. Otherwise, the execution of the additional instructions does not give any benefit. The instruction overhead is still present for codes which experience cache misses, but the benefits of hidden latency outweigh the overhead. Although for some codes, locality analysis can determine when prefetch instructions should be inserted, such analysis typically requires a certain structure of the code. For instance, Mowry reports [13] that some while-loops do not have an induction variable that the locality analysis needs.[1] A more serious limitation is that both the cache size and the input data set size (or loop iteration count) must be known at compile-time (otherwise, compile-time analysis cannot know whether cache misses will occur). Furthermore, for irregular data structures it seems difficult to determine the locality at compile-time. Therefore instruction overhead is an even more significant problem for such codes.

In contrast, hardware-based prefetch techniques execute in parallel and do not cause any instruction overhead. Stride prefetching in hardware first identifies the stride by comparing the sequence of addresses generated by each memory access instruction. Unfortunately, the stride-detection phase requires quite complex hardware which includes a reference prediction table [5]. Once the stride is detected, prefetching proceeds until hardware detects that accesses no longer use the stride. Another limitation is that no hardware-based technique in the literature has been able to prefetch indirect memory accesses.

The problems with the instruction overhead and the need for knowing the cache size and data set size at compile-time for compiler-based prefetching on the one hand, and the complex stride-detection phase and the limitation to regular accesses for hardware-based prefetching on the other, 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 new functional unit which issues prefetch requests. Our approach uses low-overhead cache miss traps as proposed in [10]. For certain memory ac-

[1] Cf. page 101 in [13].

cess 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.

For stride array accesses, the miss trap handler starts a functional unit called a *prefetch engine* by specifying prefetch parameters including prefetch start address, stride, number of prefetches to issue, direct or indirect addressing mode, and shared or exclusive mode requests. The parameters are derived statically by the compiler but the prefetch engine executes on its own and does not cause additional instruction overhead beyond that in the trap handler. Trap handlers typically complete in 40 to 50 clock cycles, which often is before the missing data arrives. The prefetch issue pace is controlled by keeping track of when a prefetched block is accessed. The number of prefetched but not yet accessed blocks is limited to a small number. There can be several prefetch engines in an implementation.

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. The emulated prefetch engine requires very little hardware support: cache miss traps and prefetch instructions. However, the number of prefetches issued by the emulated engine must be limited to a small number, otherwise hot-spots can be created. Therefore, the emulated engine cannot remove as many cache misses as can the real prefetch engine.

To evaluate our techniques we have implemented the functionality to generate cache miss traps and 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. The details of the compiler algorithm are presented in Appendix A. 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

This section illustrates the steps our technique takes from experiencing a cache miss, executing a trap handler, to starting a prefetch engine (or emulating the engine). 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. In Section 3 we will consider the hardware support our approach assumes.

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. A key difference between ordinary hardware trap handlers and our cache miss trap handlers is that the

former executes in separate stack frames. Instead, to be able to access the local variables (which may reside in registers), a cache miss trap handler is part of a faulting instruction's procedure.

The compiler controls a prefetch engine by specifiying the prefetch parameters shown in Table 1, which also shows the default value of each parameter. A trap handler needs only specify a parameter value if it is different from the default value. The parameters are as follows: *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. The cache state is normally shared but is set to exclusive mode for data that will be modified, which is useful for ownership-based cache coherence protocols [13]. The cache state is specified as two boolean values: the first refers to direct addressing and the second to indirect addressing. For instance, (false, true) specifies direct addressing prefetch using shared state and prefetch for writing using indirect addressing. So, the task of the compiler algorithm is to extract these parameters for each stride access in a loop. How this is done will be described in Section 4.

Parameter	**Default Value**	**Description**
Count	Blocks in a page	Number of blocks to prefetch
Stride	Cache block size	Distance in bytes between prefetched blocks
Indirect	Zero	Number of blocks to prefetch using indirect addressing
Cache state	Shared state	Prefetch for reading or writing
Address		Prefetch start address

Table 1: Prefetch parameters that the compiler extracts from the code. A prefetch engine uses the default value for parameters not specified.

To illustrate the operation of a prefetch engine, we will use C code fragments and show how the engine should be controlled for each case. 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 accesses one array.

To start with 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`, `n`. The index `i` is incremented by `D` in each loop iteration. 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 one array-element is E, the stride S of the instruction

loading `a[K*i]` then becomes $K * D * E$. 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 $N = \lceil (n - i) * S/(D * B) \rceil$ (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. For this code, *Stride* is set to the size of a pointer to an `int` and *Indirect* is set to one, since the size of an `int` does not exceed one cache block. If a pointed-to variable is *IB* blocks, *Indirect* is set to *IB*. Strides less than a cache block only make sense when using indirect addressing. *Count* is set to $n - i - 1$, and the start address to `&a[i+1]`. The prefetch engine is likely to find that the first block is already requested by the instruction loading `a[i]`, however, when that block arrives to the second-level cache, the prefetch engine can issue a prefetch for `a[i+1]` — i.e. the address of a pointed-to variable.

```
int     *a[];

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

Figure 2: Here indirect addressing is used.

So far, we have shown codes with only one array accessed in a loop iteration. In fact, the number of arrays accessed in a loop does not affect neither the compiler algorithm nor the resulting trap handlers in any way. A trap handler is only concerned with one array. Prefetching multiple arrays concurrently is supported by hardware through use of multiple prefetch engines. One engine is associated with each data stream. In our simulations, we use four prefetch engines. Arrays of records are treated in the same way as arrays of scalar variables. Consider a record size greater than the block size. Then each access to a field of a record is associated with a trap handler which will prefetch the different blocks (of different records) where that field is located.

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. Otherwise, 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. The number of engines is implementation-defined. Although the compiler does not need to know the exact number of engines from a correctness point of view, it can produce better code if it knows the exact number of engines. Before the compiler can specify the parameters shown in Table 1, 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. In our design, engines are selected in a round-robin manner by a counter. Apart from this selection, all engines operate independently of each other. To limit the number of pending prefetches, a prefetch engine records all prefetches in a prefetch buffer which will be described next.

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 prefetchs. 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 2: Prefetch buffer entry. Each engine has a buffer with four entries.

3.2 Prefetch Engine State

In Table 3 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.

State Variable	Default Value	Description
Operating		True if parameters are valid
Stride	Cache block size	Byte stride
Address		Next address to prefetch
Count	Blocks in a page	Remaining prefetches to issue
DState	Shared state	Direct addressing cache state
Pointer		Next address to prefetch indirectly
Indirect	Zero	Number of indirect prefetches
IState	Shared state	Indirect addressing cache state

Table 3: 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, e.g. 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.

In Figure 3 we see an example of indirect prefetching. An engine is prefetching an array of pointers and the pointed-to objects, and each element's and object's cache block number and cache block state are shown in the figure. A block for which there is a pending prefetch is marked with PF in Figure 3. Each pending prefetch has a prefetch buffer entry in which the prefetched block number can be seen. As we can see, both *Address* and *Pointer* advance through the array but *Address* advances faster, since *Pointer* must wait for the data to arrive before it can issue an indirect prefetch.

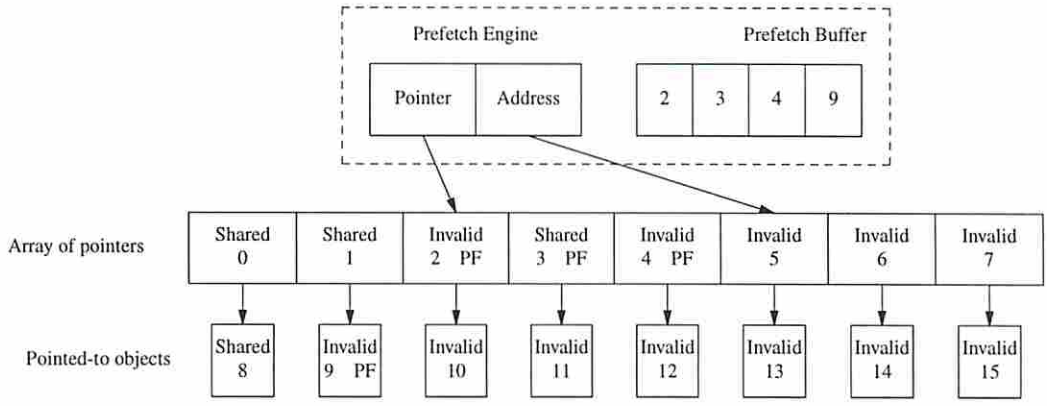

Figure 3: Indirect addressing prefetch.

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. The details of the algorithm are presented in Appendix A.

4.1 Direct Addressing Engine Prefetch

We will first consider memory accesses that can be handled by a prefetch engine using direct addressing mode. 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 $(n-i)*S/(B*D)-1$. 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 would be too time-consuming, instead we approximate the number using shift as follows:

Case 1. For $S \geq B$, we have $N = (n-i)/D - 1 = (n-i) * 2^{-log(D)} - 1$. We shift (n-i) to the right $log(D)$ positions.

Case 2. For $S < B$, we have $N = (n-i) * S/(B*D) - 1 = (n-i) * 2^{-log(B*D/S)} - 1$, so we shift (n-i) to the right $log(B*D/S)$ positions.

Case 3. For $S < B \wedge S = D$, we have $N = (n-i) * S/(B * D) - 1 = (n-i)/B - 1 = (n-i) * 2^{-log(B)} - 1$, and we shift (n-i) to the right $log(B)$ positions.

If the number of prefetches to issue cannot be computed at runtime, a default number is used instead (we use the number of cache blocks in a page).

In Figure 4, we give an example of a loop and trap handlers. Only one of *T1* and *T2* is generated, depending on the relative size of *B* and *D*. In the loop, a pointer *p* is incremented *S* bytes each loop iteration. The loop is executed while *p* is less than *u*. In this case, *S* and *D* are equal, and therefore the number of blocks to prefetch for *T1* where $S < B$ is computed using Case 3 above, while for the other case, *T2*, the number of blocks to prefetch is approximated using Case 1.

Loop	**Trap Handler if S ¡ B**	**Trap Handler if S ¿= B**
`L:`	`T1:`	`T2:`
`load p,x`	*reset*	*reset*
`add p,S,p`	`sub u,p,t1`	`sub u,p,t1`
`blt p,u,L`	`srl t1,LOG(B),t2`	`srl t1,LOG(S),t2`
	`sub t2,1,t3`	`sub t2,1,t3`
	count `t3`	*count* `t3`
	`add p,B,t4`	*stride* `S`
	address `t4`	`add p,B,t4`
	`reload saved regs`	*address* `t4`
	`rett`	`reload saved regs`
		`rett`

Figure 4: Example loop with its trap handler. Either of *T1* and *T2* is generated, which one depends on the relative size of *B* and *D*, which in this case is equal to *S*. The reloaded registers were saved by the prologue.

4.2 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. To illustrate this, consider the loop in Figure 5, where the first load instruction (A_1) is a parent and the second (A_2) is a child. The engine parameter *Indirect* is set to one because one block is fetched using indirect addressing.

A separate trap handler is not generated for A_2 (since it is not a stride access), however, if A_1 rarely misses but A_2 does frequently, we wish to do indirect prefetching to avoid these misses. Therefore, we let A_2 use the trap handler of A_1. Assume the prefetch engine in Figure 3 was initiated due to a miss to A_2; then *Address*, defined in Section 3.3, will advance quickly through the array while *Pointer* will issue prefetches. In the loop, a pointer *p* is incremented by four bytes each loop iteration. The loop is executed while *p* is less than *u*. The stride is set to four, and the number of prefetches using direct addressing is set to $(u-p)/4 - 1$.

Loop			Trap Handler (4 ¡ B)
L:			T:
A1:	load	p,c	*reset*
A2:	load	c,x	*indirect* 1
	add	p,4,p	sub u,p,t1
	blt	p,u,L	srl t1,2,t2
			sub t2,1,t3
			count t3
			stride 4
			add p,4,t4
			address t4
			reload saved regs
			rett

Figure 5: Example loop with pointer dereference and its trap handler.

4.3 Finding a Trap Handler at Runtime

For each memory access with a trap handler, a pair of two symbols are added as data to a special section in the relocatable object file: one referring to the instruction address of the faulting memory access and the other referring to the trap handler's first instruction. The link-editor relocates the symbols and stores each pair in the executable file in a hash table at the end of the text segment. At a cache miss trap, the prologue first saves a small number of general purpose registers and then searches the hash table to find the pair and then jumps to the trap handler.

5 Experimental Methodology

To evaluate our prefetch technique, we have incorporated the prefetch algorithm in an optimising compiler [17], including the modifications of the link-editor. 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. Next we define metrics of detection efficiency used when we analysed the application executions. Finally 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. Even though our compiler performs many standard optimisations, it becomes important to understand how the results compare to code compiled with other compilers. We have compared some key parameters with gcc (version 2.1) with optimisation level O2. We have found that the numbers of loads and stores to shared-memory typically differ by less than 1% between our compiler and gcc.

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 4.

Table 4: Benchmark Programs, Data Set Sizes used.

Benchmark	Description and Data Sets
Water	N-body water molecular dynamics simulation 244 molecules, 3 time steps
Cholesky	Cholesky factorisation of a sparse matrix matrix bcsstk14
MP3D	3D particle-based wind-tunnel simulator 50,000 particles, 5 time steps
LU	LU-decomposition of a dense matrix 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. We define the *coverage* to be $C = (H + P)/(H + P + M)$.

Table 5: Metrics of detection efficiency.

Metric	Description
Coverage	Fraction of completely or partially hidden memory access latencies
Bad	Fraction of prefetch requests that are useless
Prefetch activity	Number of prefetches divided by misses in baseline

Traditionally, prefetch instructions are useless if the data to prefetch is already present in the cache. In our technique, prefetch instruction overhead is not a concern and in the efficiency measurements, we count only prefetches that miss and will generate memory requests to fetch data. Thus, in this study, prefetches are useless if they do not remove cache misses. There are different reasons for a prefetch request to be useless. The requested data may arrive but be replaced or invalidated before it is accessed or the processor will never access that data even if the cache were infinite. Another category of useless prefetch requests are those which were denied data by the coherence protocol, which may happen for example when the requested block is in a transient state in memory, e.g. waiting for an update or invalidation acknowledgement from a remote

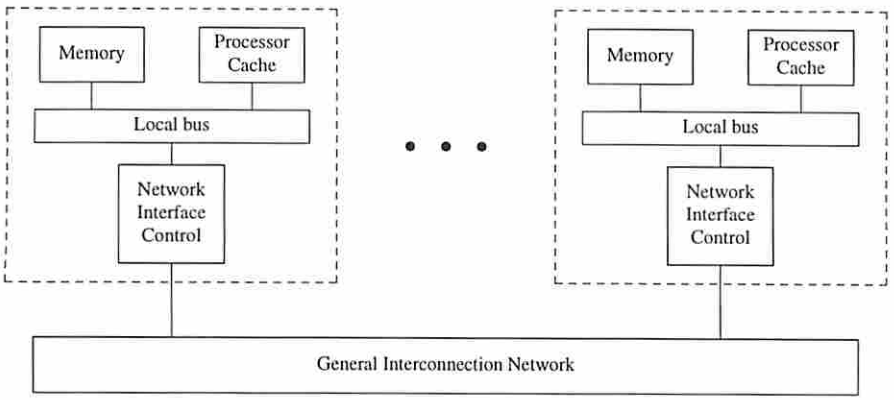

Figure 6: Organisation of a cache coherent multiprocessor.

cache. We call all such prefetches *bad* and we define the *degree of bad prefetches* to be the number of bad prefetch requests divided by the number of prefetch requests sent to memory.

Finally, we define the *prefetch activity* as the number of prefetch requests in one execution divided by the number of misses in the original execution. This metric is useful when interpreting the coverage and degree of bad prefetches: a low coverage and a high degree of bad prefetches may have no significant effect on performance if the prefetch activity is low. We summarise the metrics in Table 5.

The measurements of prefetch activity, coverage, bad, 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 Multiprocessor 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 6. 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 7. 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*) (which is also direct-mapped), and there is full inclusion between the *FLC* and the *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.

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 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 7, 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.

Support for 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* (and therefore in the *SLC* as well due to inclusion), 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 (that miss would not generate a second trap, however). A cache miss trap is a low-overhead trap [9]. In particular, the operating system is not involved in the trap, since this would be too costly in terms of instruction overhead. The additional state that faulting memory accesses require is a flag that can disable cache miss traps, so that cache miss trap handlers do not need to worry about nested cache miss traps. Cache miss traps are enabled again after a faulting memory access is re-executed, so that repeated traps are not generated. 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

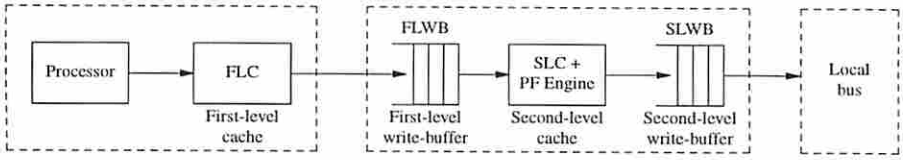

Figure 7: The two-level cache hierarchy in each processing node. The prefetch engine is part of the SLC.

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 64 kB *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

We first show the detection efficiency achieved for the prefetch engine and the emulated prefetch engine, and then as a case study present simulated execution times and traffic.

6.1 Detection Efficiency

The diagrams of Figure 8 show the coverages (top), the degrees of bad prefetches (mid), and the prefetch activity (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 in-

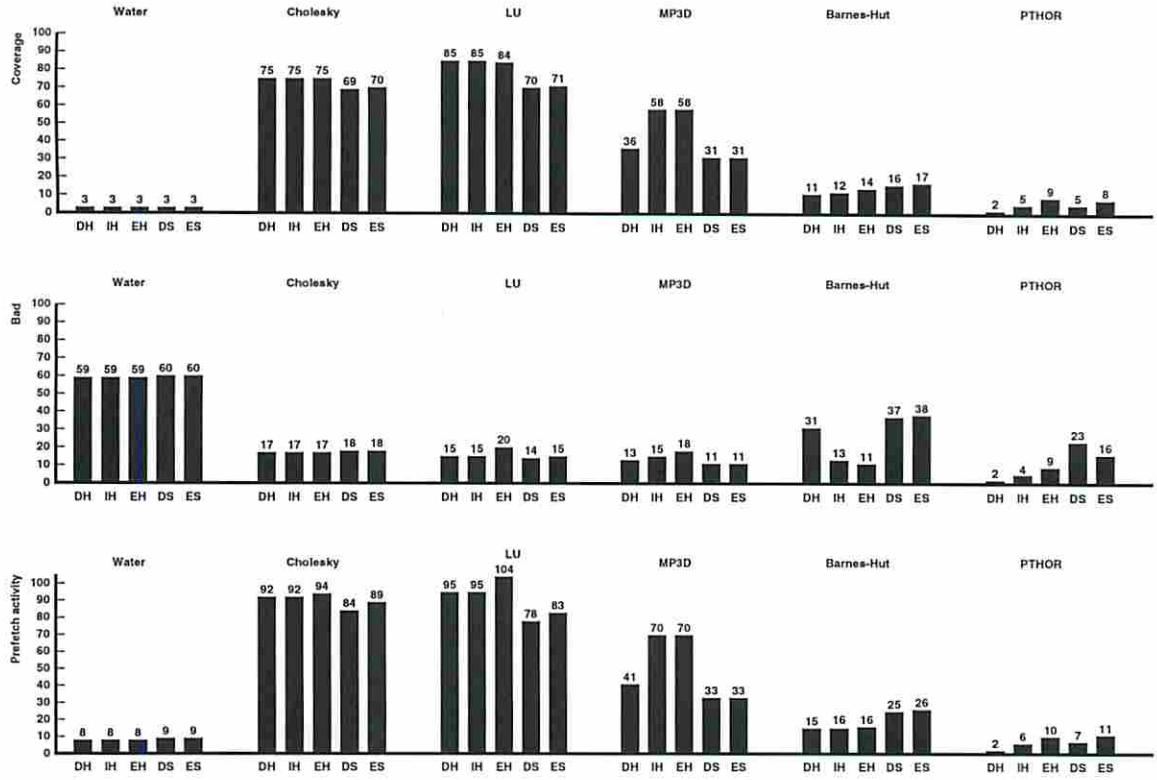

Figure 8: Coverage (top), degree of bad prefetches (mid), and prefetch activity (bottom) in percent.

direct 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.

To start with Water, we see that the number of prefetches issued is quite low for each technique. Of the 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 bad 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 bad 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 the prefetching activity is quite low and 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. *SW* has a higher prefetching activity than *HW*. 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. The prefetch activity is higher 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.

6.2 Effects on Execution Time

In this section we present the execution times for the applications in Figure 9. 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. 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 (whose absence in the cache generated the trap) has arrived.

Based on the detection efficiency measurements presented in the previous section, we can expect three applications with significantly reduced memory access stall time: Cholesky, LU, and MP3D. The other applications are interesting because we want to know how our prefetch approach affects performance for codes where it is unable to cut memory access stall times significantly. These applications are Water, Barnes-Hut, and PTHOR.

To start with Water, no prefetch technique has any effect on the the execution-time, which can be understood by the low prefetch activities.

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.

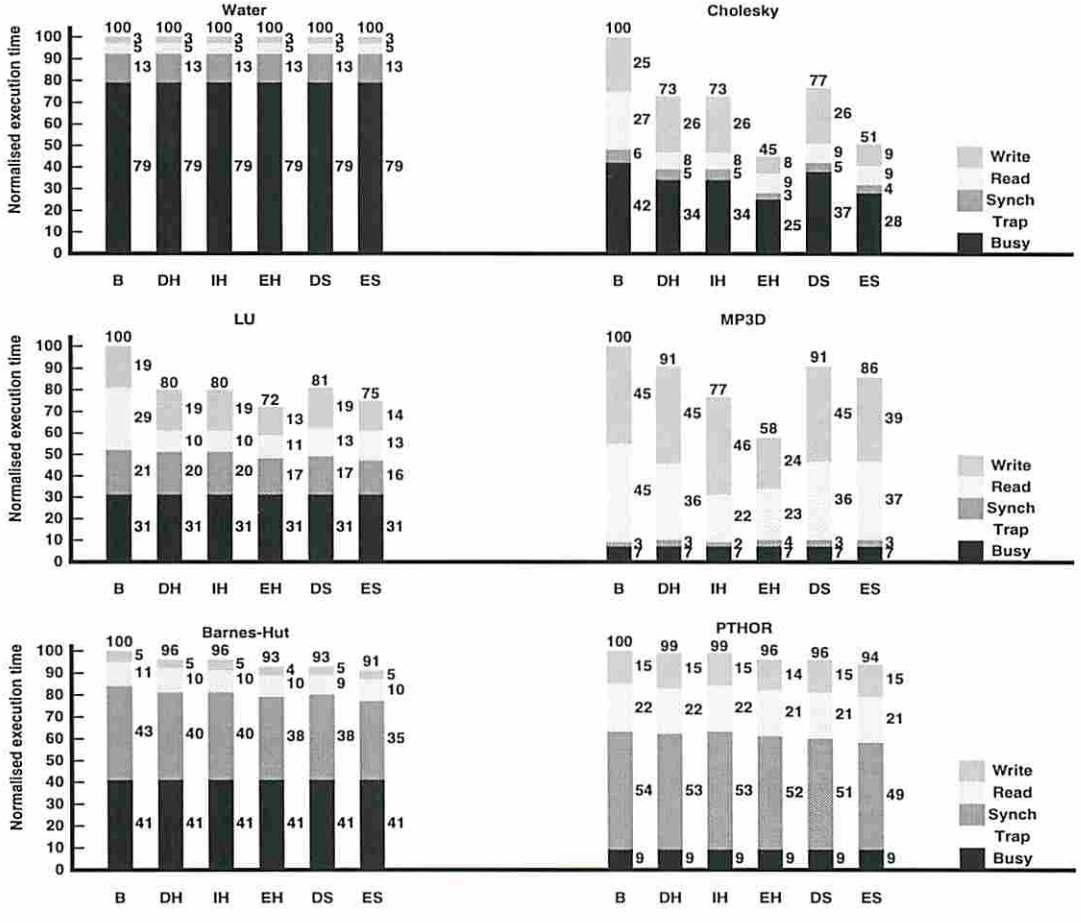

Figure 9: Normalised execution times.

For LU, *DH* and *DI* reduce the read stall time from 29% to 10%, and *EH* reduces it to 11%. As expected from the coverage measurements, *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

Table 6: Instruction Overhead due to Traps measured for *EH* and *ES*.

Benchmark	**Instruction Overhead**	
	EH	*ES*
Water	0.00%	0.00%
Cholesky	0.065%	0.11%
LU	0.059%	0.16%
MP3D	0.42%	0.0088%
Barnes-Hut	0.019%	0.030%
PTHOR	0.40%	0.41%

19% down to 13% and 14%, respectively.

For MP3D, *DH* and *SW* are not expected to reduce the read stall time significantly, as discussed in the previous section. *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. In Table 6 we can see the instruction overhead for each application measured for *EH* and *ES*. 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 can compare the instruction overhead for *EH* and *ES* only for applications without indirect engine prefetch (Water, Cholesky, and LU).

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.

6.3 Effects on Traffic

We present in this section how memory traffic is affected for each algorithm in Figure 10. A reduction in traffic comes from reducing the number of control messages while an increase is due to useless prefetches and additional cache misses created by useless invalidations. We are especially interested in seeing the effects of exclusive mode prefetching on the traffic. We can expect that the merged data and ownership requests have a potential to cut traffic, unless the ownership requests create additional misses. Recall that a control message is 5 bytes, and a data message is 21 bytes.

Starting with Water, we can see that traffic is increased by 3% for each technique except *ES* where it is increased by 4%. Thus, the high degrees of bad observed in Section 6.1 for Water do not have a significant effect on the traffic, since the prefetch activities are rather low for Water.

Continuing with Cholesky, we see that the useless prefetches increase the traffic and the merged ownership and data requests in exclusive mode prefetching reduce the traffic. For *DH* and *DS* the traffic becomes 109% and 107%, respectively, while for *EH* and *ES* it becomes less than for *B*, 96% and 98%, respectively.

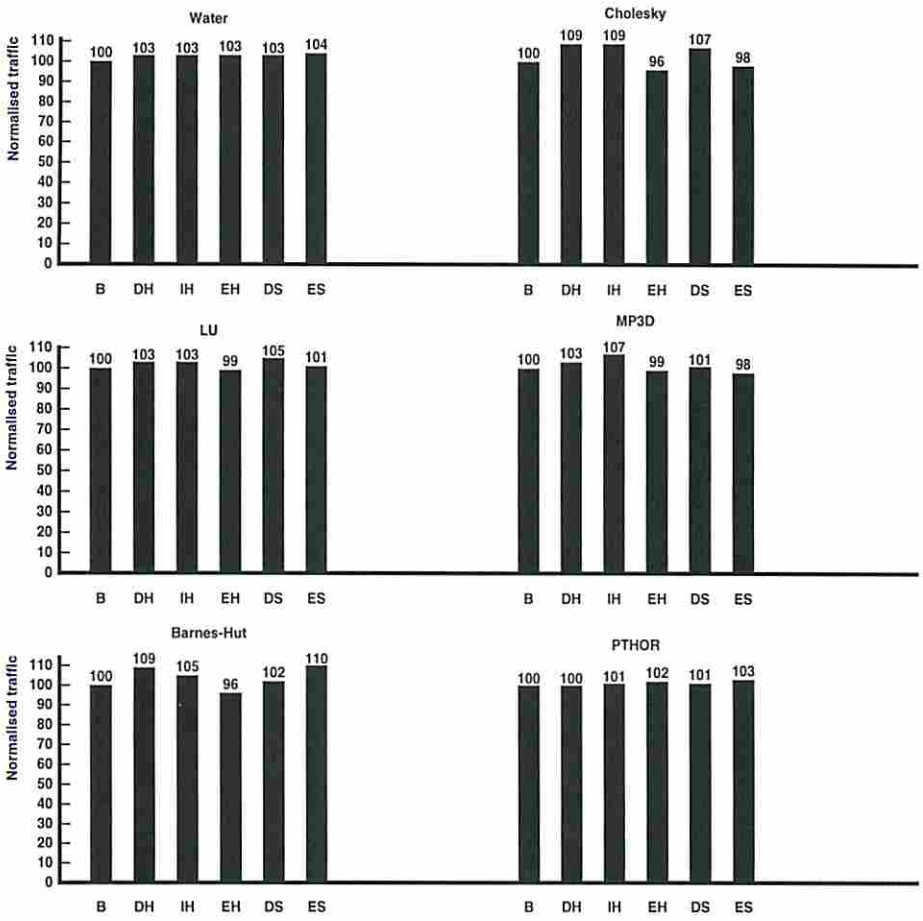

Figure 10: Normalised traffic.

The traffic generated for LU is also below or close to that of the baseline. It is as follows: *DH* 103%, *EH* 99%, *DS* 105%, and finally for *ES* it is 101%. The reason why the traffic is lower for LU than for Cholesky (even though they have similar degrees of bad) is that for LU it was control messages that were useless (nacked prefetch requests) while for Cholesky it was both control and data messages.

MP3D has a low traffic as well. We can see that exclusive mode prefetch reduces the traffic to or below that of *B*, while *IH* suffers from useless prefetches and increases the traffic to 107%.

Finally, for Barnes-Hut and PTHOR the traffic was increased somewhat. While *EH* can remove ownership requests, the reduction for Barnes-Hut is mainly due to dynamic application scheduling effects.

In summary, for all applications, the prefetch engine increased the traffic by less than 10%. We also see that exclusive mode prefetching can reduce traffic, as compared to the other schemes we evaluated.

7 Related Work and Discussion

Compiler-based data prefetching has been studied extensively by Mowry in [13]. As mentioned in Section 1, there are some problems with traditional compiler-based data prefetching as proposed in [13], and which this research aims at overcoming.

By inserting prefetch instructions in the code, these instructions will introduce overhead, regardless of whether cache hits or misses occur. In our approach, there is instruction overhead 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 limit the use of data prefetching. Since the instruction overhead is present in Mowry's approach also when there are no cache misses, that approach is limited to source codes, where the compiler can find that misses will occur. To find this, the compiler must have more information known at compile-time than our approach needs, including the data set size (or loop iteration count) and the cache size. This excludes many codes. For instance, it seems difficult to use traditional compiler-based prefetching in numerical libraries, or in object-oriented programming with precompiled classes.

In [9] Horowitz *et al.* propose a way to reduce the instruction overhead by chosing between alternate code versions at runtime; different code versions are optimised for different prefetching strategies. We will refer to the prefetching approaches in [13] and [9] as Mowry's prefetching approaches. One of their experiments is focused on the library routine `bcopy()`, which copies a block of data from one location to another. This code is an example of when locality analysis cannot be performed (since the loop iteration count is unknown at compile-time). Their experiment shows results for cases when the data fits in the cache and when it does not.

For the case when data fits in the cache, the traditional compiler-based prefetching creates significant instruction overhead of more than 80% of the original busy time. To avoid this instruction overhead, the authors suggest using informing loads to measure the prefetch *hit* count for the first few loop iterations of bcopy. If the hit count is above a certain number, the remaining loop iterations of bcopy use code without prefetch instructions. This reduced the instruction overhead to around 15%. For cases where the data does not fit in the cache, the code with prefetch instructions will be used for all loop iterations. The instruction overhead then becomes around 77%. These examples illustrate that significant instruction overhead can remain, and the purpose of our research is to remove that by using the prefetch engine. In our approach, neither the loop iteration count nor the cache size must be known at compile-time.

The reason why instruction overhead remains in the executions of their experiments is (of course) that the prefetch instructions are mixed with other instructions. In contrast, to emulate the prefetch engines, we execute prefetch instructions in a loop in a trap handler while the processor is stalled anyway. Which software prefetch approach is most advantageous depends on several factors: Firstly, the coverage can be made higher in Mowry's approaches than in the emulated prefetch engines, because an emulated engine issues only a limited number of prefetch requests while the trap handler is executing. However, as the number of prefetches issued in each trap handler invocation increases, so does the coverage. With N issued prefetches in a software loop, the upper bound of the coverage becomes $N/(N+1)$; e.g. with $N = 16$, the upper bound becomes a coverage of 94%. The risk with this amount of clustered prefetches is that hot-spots can be created. A second difference is that Mowry's approaches cover indirect addressing while our emulated prefetch engines currently do not. However, we are investigating approaches to deal with that case as well. Also, in our approach we cannot avoid the initial miss in a loop since that miss is used to start prefetching, while in [13] prefetch instructions are inserted before loops as

well (to cover initial misses). So the bottom line is that in Mowry's approach, the coverage can be higher while for the emulated prefetch engines, the instruction overhead is much smaller.

We see two main options for doing indirect prefetch using an emulated prefetch engine. Firstly, with hardware support in the cache, we could specify a prefetch instruction that takes the address of a pointer as a parameter, then both the pointer and what it points to can be prefetched. We call these indirect prefetch instructions. Secondly, we can synthesise the indirect prefetch instruction using a load and an ordinary prefetch instruction in the trap handler. In a processor with blocking load instructions, such synthesised indirect prefetches can increase read stall time, if a cache miss is suffered when reading a pointer and the pointed-to data is prefetched but not used.

In the initial stages of this research, we used faulting memory instructions for every shared memory access with the purpose of using prefetch instructions for nonstride memory accesses as well. While the coverages reached were somewhat higher, the cost of finding and executing the trap handlers sometimes exceeded the benefits of hidden latency. In contrast, restricting the use of faulting memory access instructions to stride accesses reduces the number of trap handler invocations and therefore the trap handler instruction overhead.

To contrast Mowry's approach to the real prefetch engines, the primary limitation of the prefetch engines is that they cannot cover initial misses. However, with large data set sizes, the prefetch engine will be programmed to issue a large number of prefetches and therefore the initial misses will be unimportant. On the other hand, with small data set sizes, the fraction of misses that are initial will be higher, but the total number of misses will be lower (since more data will fit in the caches) and therefore prefetch may not be very important. Based on this discussion, be expect that the prefetch engines can reach the same performance improvements as Mowry's approach. A more important difference, however, and which is the difference which motivates our approach, is that Mowry's approach is only applicable to codes where compile-time analysis can determine that there will be misses, while our approach frees the compiler from this concern.

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. Since the latency of a first-level cache-miss serviced by the second-level cache is much shorter than the 40-50 clock cycles that our trap-handlers need to terminate, there is a risk that the trap-handlers create instruction overhead which exceeds the benefits of prefetching. It would be interesting to explore this further.

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 new functional unit 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 trap handlers are part of the procedure of the trapping instruction and therefore can access the procedure's local variables. From the local variables, the trap handler can compute the remaining number of cache blocks of an array that the processor will access in a loop. The prefetch engine is initialised 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 have presented a detailed evaluation of our prefetch technique including effects on execution time and traffic. 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 are planning to evaluate the prefetch engine in the context of a superscalar microprocessor. 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.

A. Compiler Algorithm Implementation

In this appendix we give a detailed presentation of the algorithm for generating trap handlers. To start with, the prefetching algorithm performs better with interprocedural dataflow information, however, our compiler currently performs no interprocedural dataflow analysis (other than that described in this section). We therefore use procedure integration to avoid introducing an artificial limitation to the prefetch algorithm. For recursive procedures, however, the prefetch algorithm exploits interprocedural dataflow analysis of access classes, as we will see in this section.

The analysis of the prefetch algorithm always operates on the flow graph of one procedure. The algorithm consists of the following seven passes:

1. **Natural loop analysis.** This pass finds the natural loops in a flow graph.

2. **Collect access classes.** This pass collects all access classes that exist in the flow graph.
3. **Local analysis.** This pass operates on each basic block and constructs dataflow information.
4. **Identifying parent and child instructions.** This pass collects information for indirect prefetching.
5. **Dataflow analysis of access classes.** This pass performs iterative dataflow analysis of access classes.
6. **Engine prefetch.** This pass operates on each natural loop, starting with the innermost loops.
7. **Procedure parameter analysis.** This pass associates access classes with the formal parameters of a procedure.

Natural Loop Analysis

The purpose of this pass is to identify the set of natural loops in a flow graph. A natural loop is a set of basic blocks with one basic block denoted the *loop header*. The structure of a natural loop is such that the loop header is the single entry point to the loop and there is at least one path back to the header [1].

Collect Access Classes

The purpose of this pass is to identify all access classes that exist in the flow graph.

The data address of a memory access is given by a base pointer b and an immediate-valued offset k. We classify all memory accesses in a procedure into equivalence classes called *access classes* such that a memory access instruction corresponds to the access class $(b, n = \lfloor k/B \rfloor)$, where B is the cache block size. Dataflow analysis of access classes is performed in the backward direction. A memory access is said to *generate* an access class and an assignment to a base b is said to *kill* all access classes with b as base. The dataflow analysis is similar to live-variables analysis and if an access class has propagated backward to a point in the flow graph, it is said to be *live* at that point. The access class $(b, n + 1)$ is called the *successor* of (b, n). When different word offsets are used to generate an access class (b, n), then two cache blocks may be accessed (since access classes are not necessarily aligned on cache block boundaries). To deal with this problem we generate the successor access class $(b, n + 1)$ when the algorithm detects that different words generate an access class (b, n). characterised by a base pointer and an offset.

A new access class is inserted by assigning it an index to simplify dataflow operations. Each base pointer is associated with a number of indices corresponding to access classes that use this base pointer. When the content of the base pointer is changed, all corresponding access classes can then be killed by simply performing bit operations on the bit vectors used in the dataflow analysis described next.

Local Analysis

The purpose of this pass is to construct, for each basic block, three sets of access classes called *GEN*, *KILL*, and *S_GEN*, which are used by the global dataflow analyses. *GEN* represents generated access classes, *KILL* represents killed access classes, and *S_GEN* represents access classes

generated by store instructions. The prefix *S* of a dataflow set means that the access classes were generated by store instructions. The set of access classes that are live at the beginning of a basic block are denoted *IN* and *S_IN*, respectively, and the access classes that are live at the end of a basic block are denoted *OUT* and *S_OUT*, respectively.

Assume *B* is a basic block with three local dataflow sets *GEN*, *KILL*, and *S_GEN*. Moreover, there is a function—denoted *access_classes*(*b*)— which maps a base pointer *b* to the set of access classes that use *b* as a base pointer. The statements of a basic block are scanned in the backward direction. When a statement is a memory access with base pointer b and offset k, the access belongs to the access class $(b, n = \lfloor k/B \rfloor)$, where B is the cache block size. If (b, n) belongs to *GEN* but was generated by an offset other than k, then multiple words of (b, n) are accessed, and the successor $(b, n+1)$ is added to *GEN* and removed from *KILL*. (b, n) is generated and added to the *GEN* set and removed from the *KILL* set:

$$B.GEN := B.GEN \bigcup \{ (b, n) \}$$

$$B.KILL := B.KILL - \{ (b, n) \}$$

If the statement is a *store*, the access class is added to *S_GEN* as well:

$$B.S\_GEN := B.S\_GEN \bigcup \{ (b, n) \}$$

Similarly as for *GEN*, the successor $(b, n+1)$ may be added to *S_GEN*.

When the statement is a function call each formal parameter *p* of the callee *c* is considered. The access classes of *p* that belonged to *IN* and *S_IN* of the initial basic block of *c* are added to *GEN* and *S_GEN* and removed from *KILL* of B. Before they are added, however, the base is translated from the formal parameter in the callee to a base that is the argument in the caller. When the statement is an *assignment of a base pointer*, all access classes for that base are killed. This is done by removing them from the *GEN* sets and adding them to the *KILL* sets:

$$B.GEN := B.GEN - access\_classes(b)$$

$$B.S\_GEN := B.S\_GEN - access\_classes(b)$$

$$B.KILL := B.KILL \bigcup access\_classes(b)$$

Identifying Parent and Child Instructions

The purpose of this pass is to identify parent load instructions which read data that is used as a base in other memory access instructions.

Dataflow Analysis of Access Classes

The purpose of the dataflow analyses is to keep track of which access classes are live at the beginning and at the end of each basic block by letting access classes generated in one basic block propagate backward to the preceding basic blocks. Note that the global dataflow information is not used by access class prefetching (which only considers local dataflow information to limit the size of the trap handlers). The global dataflow information is used for engine prefetch.

For all immediate successor basic blocks S of B in the flow graph, we can formulate how access classes that are live at the beginning of each S propagate to B. This is done in the following dataflow equations:

$$B.OUT := \bigcup S.IN \quad \forall \text{ immediate successors } S \text{ of } B$$

$$B.S\_OUT := \bigcap S.S\_IN \quad \forall \text{ immediate successors } S \text{ of } B$$

Thus, for prefetching for reading, the algorithm requires only that an access class is live in at least one successor basic block. In contrast, for prefetching for writing, it requires that an access class is live at the beginning of all immediate successor basic blocks.

Finally, the dataflow equations to establish $B.IN$ and $B.S\_IN$ are as follows:

$$B.IN := (B.OUT \bigcup B.GEN) - B.KILL$$

$$B.S\_IN := (B.S\_OUT \bigcup B.S\_GEN) - B.KILL$$

i.e., access classes live at the beginning of B are defined by the access classes live at the end of B plus access classes generated in B minus access classes that are killed locally in B. The above operations applied to each basic block are repeated until no changes occur as in other iterative dataflow analyses [1].

Engine Prefetch

The purpose of this pass is to generate cache miss trap handlers for controlling prefetch engines. Before this pass, all natural loops of a procedure have been sorted from innermost to outermost loop. The loops are processed starting with the innermost. Each natural loop *L* of the flow graph is considered in turn. A stride access is an access whose address-expression contains a term which is an induction variable *IV* of *L*. *IV* is incremented by a compile-time constant *c* each loop iteration. Each stride access *A* in *L* is considered in turn, and is then marked as visited. A stride access is visited only once— in the innermost loop where there is an induction variable in the access's address expression.

First, the stride is computed at compile-time by analysing the term with *IV* of the address-expression of *A*. The stride *S* of *A* is set to *c*, possibly multiplied by a constant in the address-expression.

Second, the compiler tries to extract the remaining number of loop iterations. If the loop header of *L* has a conditional branch instruction of the form $i < n$ where i is an induction variable of *L*, then *L* is said to be *limited*. Note that *IV* and *i* can be different variables. If *L* is not limited, then the remaining number of loop iterations is not estimated and therefore not the number of blocks to prefetch either, however, prefetching is used anyway but the number of blocks to prefetch is left unspecified and a default value is used by the prefetch engine; for the emulated prefetch engine, the number of blocks to prefetch is set to a small number (we used four).

If *L* is limited, then the induction variable i is incremented by a constant D in each iteration. In the loop of Figure 1, the loop is limited and the induction variable is i. The remaining number of loop iterations is estimated to be $N = (n - i)/D$, where n and i are runtime variables. Early loop exits due to a `goto` or a `break` or additional loop termination conditions beyond $i < n$ are not taken into account by our algorithm. If the stride *S* of *A* is greater than or equal to one cache block *B*, then the number of blocks to prefetch is set to one less than N (as discussed in Section 3, the missing access requests the first cache block by itself). However, if *S* is less than *B*, then multiple loop iterations will access the same cache block, and number of blocks to prefetch is set to $N * S/B - 1$.

If A is a store instruction or the access class of A is either generated by a store in the same basic block B or belongs to *B.S_OUT*, then the engine will prefetch in exclusive mode under direct addressing.

If A is a parent, then indirect prefetching will be used. The engine parameter for *Indirect* is determined by considering which access classes of a child c are live at the point just after A in the flow graph. The number of blocks to prefetch using indirect addressing and the requested cache state (shared or exclusive) for these blocks are determined as follows. First, the number of consecutive access classes $(c, 0), (c, 1), ..., (c, r)$ that are live at the point just after A in the flow graph is found. Second, the number of consecutive access classes generated by store instructions $(c, 0), (c, 1), ..., (c, w)$ that are live at the point just after A in the flow graph is found. We have $w \leq r$. If w is greater than zero, then exclusive mode prefetching is used for indirect prefetch and the number of blocks to prefetch using indirect addressing is set to w. Otherwise, if r is greater than zero, then the number of blocks to prefetch using indirect addressing is set to r and the blocks are prefetched for reading only.

Finally, a trap handler is created and is added to the flow graph.

Parameter Analysis

Each formal parameter p a of procedure is finally considered. All access classes (p, m) that belong to *IN* and *S_IN* of the initial basic block of the flow graph are preserved in the symbol table as interprocedural dataflow information of access classes. If the procedure is recursive, then all analysis is performed a second time; this time exploiting the formal parameters of itself in the local analysis.