Assisted Execution

Michel Dubois and Yong Ho Song

CENG Technical Report 98-25

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

October 1998

Assisted Execution

Michel Dubois and Yong Ho Song

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213)740-4475
Fax: (213) 740-7290
{dubois, yongho}@paris.usc.edu

Abstract

We introduce a new execution paradigm called *assisted execution*. In this model, a set of auxiliary "assistant" threads, called *nanothreads*, is attached to each thread of an application. Nanothreads are very lightweight threads which run on the same processor as the main (application) thread and help execute the main thread as fast as possible. Nanothreads exploit resources that are idled in the processor because of dependencies and memory access delays.

Assisted execution has the potential to alter the current trade-offs between static and dynamic execution mechanisms. Nanothreads can monitor and reconfigure the underlying hardware, can emulate hardware and can profile applications with little or no interference to improve the program on-line or off-line.

We demonstrate the power of assisted execution with an important application, namely data prefetching to fight the memory wall problem. Simulation results on several SPEC95 benchmarks show that sequential and stride prefetching implemented with nanothread technology performs just as well as ideal hardware prefetchers.

Keywords: Multithreading, prefetching, ILP processors

Assisted Execution

Abstract

We introduce a new execution paradigm called *assisted execution*. In this model, a set of auxiliary "assistant" threads, called *nanothreads*, is attached to each thread of an application. Nanothreads are very lightweight threads which run on the same processor as the main (application) thread and help execute the main thread as fast as possible. Nanothreads exploit resources that are idled in the processor because of dependencies and memory access delays.

Assisted execution has the potential to alter the current trade-offs between static and dynamic execution mechanisms. Nanothreads can monitor and reconfigure the underlying hardware, can emulate hardware and can profile applications with little or no interference to improve the program on-line or off-line.

We demonstrate the power of assisted execution with an important application, namely data prefetching to fight the memory wall problem. Simulation results on several SPEC95 benchmarks show that sequential and stride prefetching implemented with nanothread technology performs just as well as ideal hardware prefetchers.

1. Introduction

Dynamically-scheduled superscalar processors exploit instruction-level parallelism (ILP) to speed-up the execution of programs. However, because of control and data dependencies and memory access penalties, large amounts of hardware and compiling efforts reap small performance gains, often resulting in vast underutilization of the hardware. Processor multithreading, and more specifically *simultaneous multithreading*, is a very promising approach to deal with these technological trends. In this approach, several threads are scheduled at the same time and compete for issue slots in the processor, reducing the impact of control and data dependencies in each thread on the CPI [23][24].

When the threads belong to independent tasks each thread may run slower because of resource conflicts with its peers; for example, running more independent threads on the same processor leads to more cache misses and memory latency to hide, which, in turns, calls for more threads. The threads may also be part of the same application, in which case the application runs faster. However, if a compiler decomposes an application into N concurrent threads and if each processor needs k threads to run efficiently, then the number of useful processors in a multiprocessor configuration is limited to N/k.

Another approach to exploit the abundant hardware resources of a multithreaded processor is to create more work to facilitate and accelerate the execution of each application thread. This extra work is executed by a set of auxiliary "assistant" threads called *nanothreads* and attached to each application thread. Nanothreads are very lightweight threads which run on the same processor as the main (application) thread, share its memory and may share its registers and its execution stack. Nanothreads exploit resources that are idled in the processor because of dependencies and memory access delays. This new execution paradigm is called *assisted execution*.

Under assisted execution, the compiler or programmer can create nanothread code customized to the dynamic properties of application programs. Nanothreads can monitor and reconfigure the underlying hardware, can emulate hardware and can profile applications with little or no interference to improve the program on-line or off-line. In a nutshell, assisted execution has the potential to alter the current trade-offs between static and dynamic execution mechanisms.

In this paper, we develop the concept of assisted execution and apply it to an important problem: attacking the memory wall problem by stride and sequential prefetching. We first introduce the execution model, and describe a possible architecture to support it in Section 2. Sections 3 and 4 elaborates on the architecture simulation model, and the sequential and stride prefetching mechanisms. The experimental methodology is given in Section 5. The simulation results comparing nanothread-based and ideal hardware prefetchers are then presented and discussed in Section 6. Finally, we review related work and conclude in Sections 7 and 8.

2. Execution Model and Architecture for Assisted Execution

2.1. Execution Model

We attach a collection of *nanothreads* to each main (application) thread. The main thread carries the computation as in a traditional environment, while the nanothreads do all the work necessary to monitor the main thread execution and possibly effect changes to improve its performance. A main thread and its nanothreads run on the same processor. Nanothreads can be started by main thread code or, as we will see, by *nanotraps*, a lightweight trapping mechanism. They interact with the main thread by sharing memory, and possibly registers and execution stack.

Figure 1. Decomposition of a Process into Threads and Nanothreads

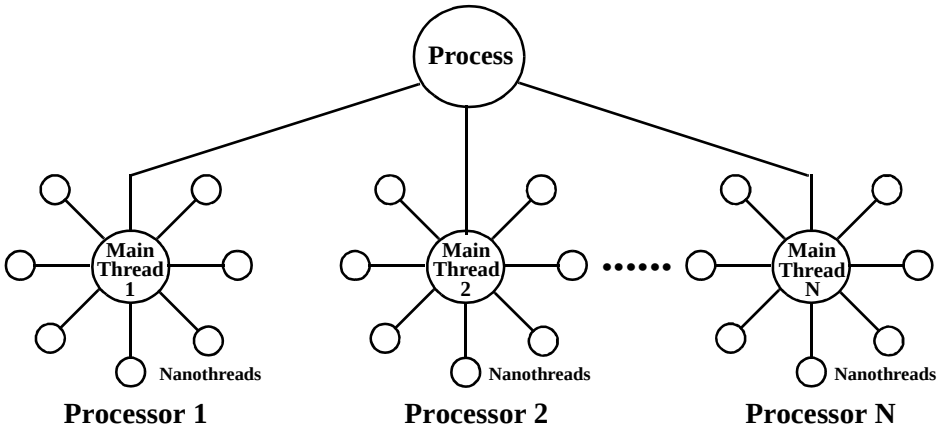
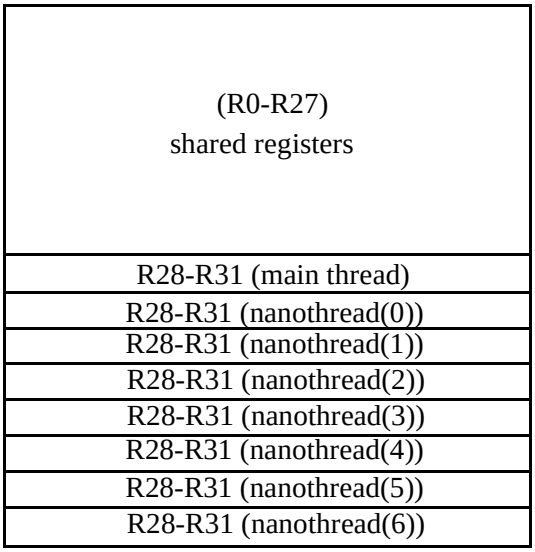


Figure 1 illustrates the case where an application has been decomposed into N main threads running on N processors and seven nanothreads are attached to each thread. Whereas the figure seems to imply an homogeneous system, nothing prevents programmers to design nanothreads customized to different compute nodes in an heterogeneous system.

In conventional multithreading [21], an active thread is characterized by the state of its register file, its program counter, and its execution stack. Since we want the main thread and its nanothreads to interact at the lowest possible level, while allowing them to execute different instructions, they should be able to share memory, registers and execution stack, but should have independent program counters. A possible organization for the integer register file is shown in Figure 2 for a processor supporting 7 nanothreads. The threads have access to 32 general-purpose registers; however 4 of these registers are private (registers 28 to 31), while 28 are shared (registers 0 to 27). Thus the total number of integer registers is 60. This organization gives some private workspace to each thread. The same organization may apply to the floating-point register file.

Figure 2. Possible Integer Register File Implementations



If the main thread and its nanothreads do not share execution stacks, they can execute unrelated pieces of code and have some level of protection from each others' execution. However, this generality comes with a cost in increased complexity in terms of synchronization and sharing registers across contexts. On the other hand, if the main thread and its nanothreads are restricted to sharing the same execution stack these issues are greatly simplified but their functionality is limited such that a nanothread can only execute within the scope of the function in which it is created. Threads sharing the same execution stack are referred to as *tightly-coupled* whereas threads having independent execution stacks are said to be *loosely-coupled*. In this paper a main thread and its nanothreads are tightly-coupled.

To illustrate the problems, consider the simple Fortran DO-loop shown in Figure 3. Assume that we create three nanothreads at the start of the loop to prefetch blocks of `A`, `B`, and `C` and that the loop index `N` is allocated to a register, which is known and shared by the three nanothreads. In Figure 3(a), tightly-coupled threads will do, since the main thread remains in the same context and the nanothreads are simple enough that they do not use function or subroutine calls. However, in Figure 3(b), a function `FUNCT()` is called in the main thread, which may use the register allocated to `N`. Thus, if the threads are tightly-coupled, the main thread must prevent its nanothreads from accessing `N` during the execution of the function. Moreover, if the prefetch algorithm is somewhat complex and requires a function call itself, sharing the same stack will cause the execution to become unpredictable, because return addresses from different nanothreads

will be mixed on the shared stack.

Figure 3. Codes Illustrating the Need for Both Tightly and Loosely-coupled Threads

```
     DO 100 N=1,200
     N_CREATE(Prefetch(A[N+1])
     N_CREATE(Prefetch(B[N+1])
     N_CREATE(Prefetch(C[N+1])
     A[N]=B[N]+C[N]
     C[N]=C[N-1]
     B[N]=A[N+1]
100  CONTINUE
```

(a)

```
     DO 100 N=1,200
     N_CREATE(Prefetch(A[N+1])
     N_CREATE(Prefetch(B[N+1])
     N_CREATE(Prefetch(C[N+1])
     A[N]=B[N]+FUNCT(C[N])
     C[N]=C[N-1]
     B[N]=A[N+1]
100  CONTINUE
```

(b)

The example of Figure 3(a) also illustrates the need to synchronize the main thread and its nanothreads. If the prefetching nanothreads are not executed fast enough, they could read the index of subsequent loop iterations. General-purpose synchronization mechanisms between the main thread and its nanothreads could include timestamps, synchronization registers, hardware flags or memory-based locks.

2.2. Processor Architecture

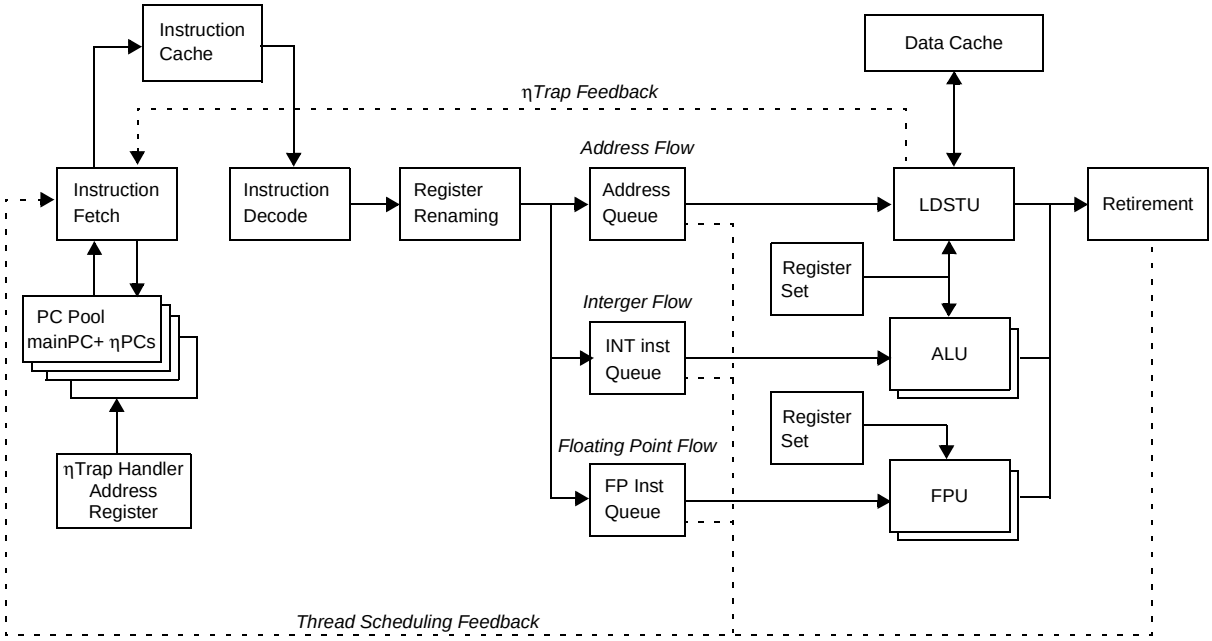
Processor architectures in which multiple threads can run concurrently can support assisted execution. For example, a microprocessor in which one nanothread can be tightly-coupled with a main thread is described in [7][1]. The processor is a VLIW (Very Long Instruction Word) processor, and, whenever the processor stalls on the main thread, it automatically begins fetching instructions from the nanothread. The modifications to the processor needed to support the nanothread is minimal.

Our focus in this paper is on dynamically scheduled, ILP processors, exemplified by the MIPS R10000 [13], shown in Figure 4 with some modifications to support assisted execution. To allow concurrent execution of one main thread and several nanothreads, the processor must have one *main PC* statically dedicated to the main thread context and several *nano-PCs*, dynamically allocated to nanothreads. In each cycle, the instruction fetch unit selects one of the active threads and fetches several instructions at a time from that thread. Multiple instructions are decoded and then sent to several instruction queues. Once its operands are available an instruction can be issued to its execution unit. Instructions finish their execution in the retirement buffer where they

1. The term *nanothread* was adopted from this paper.

wait for their turn to retire in the program order of their thread.

Figure 4. ILP Processor with Support for Nanothreads



2.3. Nanotraps

To implement precise exceptions, the MIPS R10000 identifies the faulting instruction at the retirement stage, prevents it from retiring and aborts all subsequent instructions in program order. The time taken by an exception is divided into four major components. The first is the delay from the exception event to the detection in the retirement stage. The second is the time required to save the processor context so that the trap handler can run. The third is the execution time of the exception handler, and the fourth is the time to restore the saved process context. Because the overhead of traditional exceptions is so large in ILP processors, they must occur rarely, which prevents their widespread use for dynamic execution mechanisms. Another restriction is that only one exception handler can be executed at a time.

With the support for nanothreads, we implement *nanotraps*, a form of lightweight traps. Nanotraps are triggered on selectable hardware events occurring in the main thread, such as cache misses, cache invalidations, or completion signal from an autonomous hardware machine. When a nanotrap is triggered somewhere in the processor, it is immediately taken by the hardware. The hardware selects a nano-PC (if any one is free) and allocates it to the nanotrap handler. Since the

handler runs in a nanothread, the processor does not switch context at the occurrence of a nanotrap. Rather, the main thread continues executing in the pipeline if it can, while a nanothread executes the nanotrap handler using resources not held by the main thread.

Nanotraps can be *synchronous* or *asynchronous*. In the synchronous case, the main thread is blocked until the handler is finished. An asynchronous nanotrap simply spawns a new nanothread. Whether an asynchronous nanotrap is blocking or not depends on what happens when all nano-PCs are busy at the time of the trap: It is blocking if the hardware stalls the main thread until a nano-PC is available and it is non-blocking if the nanotrap is simply ignored. Blocking asynchronous nanothreads are useful when the nanotrap cannot be ignored, for example if the nanotrap responds to the overflow of an event counter. In many cases, such as prefetching, non-blocking asynchronous nanotraps are sufficient.

To demonstrate the effectiveness of assisted execution, we have developed a simulator of a processor for assisted execution. Then we have implemented and evaluated sequential and stride prefetching schemes using prefetching nanothreads triggered by nanotraps. In the next section, we describe the specific details of the architecture simulated.

3. Detailed Architecture Model

Referring to Figure 4, we first define some terms. An instruction is *ready* for execution when all its register operands are available, either in a register or through forwarding. A ready instruction in an instruction queue *issues* when it starts execution. Arithmetic instructions are *completed* when their execution is finished in the assigned execution unit. Load instructions are *completed* when the data is returned and forwarded to the dependent instructions. The target register of an arithmetic or load instruction is not updated until the instruction *retires* in the *retirement* unit. Instructions must retire in program order. Store instructions are *completed* when they retire and update the cache.

In every cycle, the Instruction Fetch Stage (IFS) selects one PC, decides the number of instructions to fetch, fetches the instructions, and sends them to the instruction decode stage. Two simplifications have been made in our model. First, instructions always hits in the instruction cache and, second, branch prediction is perfect.

The instruction fetch scheduler selects a thread based on the number of active nanothreads and the number of instructions in the processor for each active thread. If the number of main thread instructions is more than 50% of the capacity of all instruction queues, or if the number of main thread instructions in any instruction queue is larger than 2/3 of the queue capacity, IFS selects a nanothread with the smallest number of instructions in the processor. The number of instruction fetched in each cycle is less than or equal to the number of free slots in any one instruction queue and its maximum is four.

The next stage is the Instruction Decode Stage (IDS), which decodes up to four instructions in every processor clock. When any instruction queue is full, the decoding stage stalls. The Register Renaming and Enqueue Stage (RRES) resolves both Write-After-Write (WAW) dependency and Write-After-Read (WAR) hazards by renaming registers using 128 integer registers and 128 floating point registers.

Renamed instructions are attached to one of three different instruction queues: integer, float-point and address. Each instruction queue contains up to 12 entries. Therefore, the maximum number of instructions pending for execution is 36. Up to four instructions can be sent to the instruction queues in each processor cycle.

In every cycle, up to five instructions are selected to issue to the execution units based on a greedy algorithm that gives a higher priority to the oldest, ready-to-issue instruction. The instruction issue scheduler also gives higher priority to nanothreads in the address queue in order to drain them as fast as possible and avoid deadlocks.

Integer instructions and floating-point instructions can be issued out-of-order, as soon as they are ready. As soon as its execution is completed, an arithmetic instruction releases any RAW register dependency with subsequent instructions, leaves the instruction queue, and proceeds to the retirement stage.

The address queue is managed in FCFS manner. No instruction in this queue can be issued until all instructions in front of it have been issued and have computed their memory address. Moreover a load cannot issue if a store with the same address is pending in front of it in the queue. As soon as the data is returned, loads forward the results to dependent instructions. All load/store instructions are kept in the address queue until they retire.

Instructions can be issued to each execution unit at the maximum rate of one every cycle. The execution time of each type of instruction is shown in Table 1.

Table 1: Execution Time of instructions

Instruction Class	Execution Time (in cycle)
Integer Instructions	1
Floating Point Instructions (except FDIV, FSQRT)	2
Floating Point Instructions (FDIV, FSQRT)	4
Address Instructions (FLC cache hit)	2

After their execution, instructions move to the retirement buffer where they wait their turn to retire in the program order of their thread. Registers identify the next instruction to retire in each thread. Each instruction is tagged with a thread identifier and a serial number. There is no limit on the size of the retirement buffer and the number of retiring instructions in a processor cycle.

3.1. Support for Nanotraps

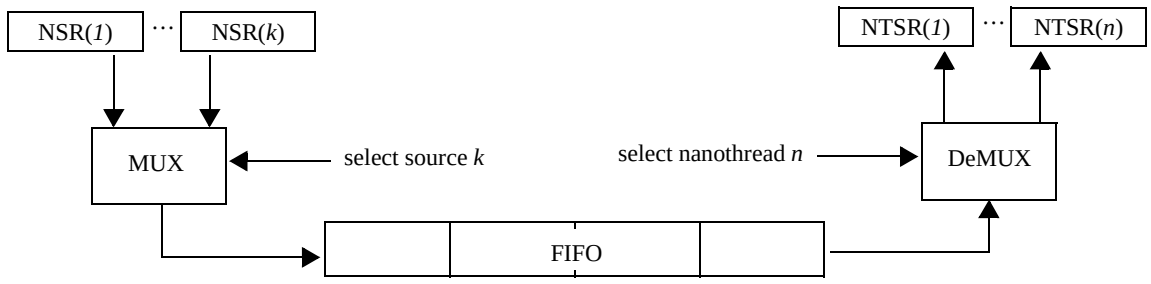
To support nanotraps a special register called Nanotrap Handler Address Register (NHAR) is added to the instruction fetch stage. NHAR keeps the start address of a common nanotrap handler. At the occurrence of a nanotrap, an inactive nano-PC (if any) is allocated to the nanothread and initialized to the content of NHAR. The nano-PC is returned to the inactive pool of PCs when a return from trap instruction (RETT) is executed in the nanothread.

Once dispatched the nanotrap handler must identify the source of the nanotrap. This is done through a set of Nanotrap Status Registers (NSR). In this paper there is only one NSR, associated with the second-level cache, but, in general we can imagine multiple NSRs, associated with other resources in the machine. Moreover, whereas at most one nanotrap may occur in every cycle in the second-level cache, multiple nanotraps could occur at the same time in the general case. We therefore need a mechanism to serialize nanotraps and to give time to the hardware to react and allocate a nanothread to each nanotrap without losing any. The mechanism is shown in Figure 5.

In every processor clock, the FIFO register is shifted forward. Whenever a new nanotrap is detected through a valid NSR, the NSR is selected. While the NSR moves up in the FIFO, a

new nano PC is selected and initialized to the value in NHAR. When the NSR emerges from the FIFO, the hardware is ready to store it in a nanothread-specific trap status register (NTSR). In the next clock the newly activated nanothread is a candidate for instruction fetch. The FIFO in Figure 5 gives the hardware four clocks to react to the nanotrap.

Figure 5. Nanotrap Status Hardware



3.2. Memory Subsystem

The memory subsystem contains separate instruction cache and data cache. The instruction cache is not simulated. This is equivalent to assuming that the instruction cache never misses. The data cache is made of a First Level Cache (FLC) and a Second Level Cache (SLC). Both are single-ported, write-back, direct-mapped with 32 byte blocks. FLC and SLC access times are 1 pclock and 6 pclocks respectively.

SLC is non-blocking. In a nonblocking cache there are two types of misses: primary and secondary misses [9]. A primary miss triggers a block fetch from memory. A secondary miss does not access memory because a primary miss for the same block is already in progress. All accesses missing in SLC occupy a slot in a Pending Memory Access Queue (PMAQ). Accesses causing secondary misses are merged with their corresponding primary miss in PMAQ. Up to 32 memory accesses (including prefetches) may be in progress in the second-level cache at any one time. The memory access latency is either 50 or 200 pclocks. Memory conflicts are not simulated.

4. Prefetching

We have experimented with both sequential and stride prefetching. Only load/store *primary* data misses in SLC may generate nanotraps. Prefetch instructions are sent to the address

queue in the processor. However, contrary to other memory access instructions, they do not wait in the queue until retirement. Rather, they are sent to the second level cache when they can be issued and are removed from the address queue. If a prefetch instruction hits in SLC, it is dropped. Otherwise, it is inserted in PMAQ as are other load/store misses.

4.1. Software sequential prefetching

In sequential prefetch, whenever a miss occurs in the second-level cache, the blocks following the missing block in the address space are *prefetched* into the cache. The number of prefetched blocks can be adjusted for different programs, but remains constant for the entire execution of a given program. Software sequential prefetch executes the same nanothread code for every miss. The NTSR must be loaded with the address of the missing access. The nanothread does not read any register of the main thread and there is no need to synchronize.

The nanotrap handler code for sequential prefetching is shown in Figure 6. By simply increasing the faulting memory address (obtained from the NTSR) by the cache block size, the trap handler generates a number of prefetches equal to `prefetch count`.

Figure 6. Software Sequential Prefetch Handler

```
prefetch_address = faulting memory address;
i = prefetch count;
do {
        prefetch_address = prefetch_address + CACHE_LINE_SIZE;
        issue prefetch from prefetch_address;
} while (--i > 0);
return from trap;
```

4.2. Software stride prefetching

Stride prefetch relies on the compiler to tag memory instructions which may trigger a prefetch on a miss and to identify the stride that should be used, as was done in [20]. Each tagged memory access instruction in the program may have its own nanotrap code. The NSTR is thus loaded with the value of the PC for the faulting instruction. The starting address of the prefetch is calculated from the context of the main thread stored in registers or on the execution stack.

Figure 7. Prologue of the Software Stride Prefetch Handler

```
pc = faulting PC;
start = start address of hash table;
index = pc & 0xfff;
hash_tbl_addr = start+index;
while (1) {
        reads a hash table entry from hash_tbl_addr;
        if (pc == address defined at hash_tbl_addr) goto trap_start_addr;
        if (the entry is blank) break;
        increase hash_tbl_addr;
}
return from trap;
```

The software stride prefetching handler is preceded by a *prologue*, shown in Figure 7 which looks up a hash table using the PC value found in NTSR. If there is a valid entry in the hash table, the faulting address qualifies for prefetching and the handler jumps to the code corresponding to the specific PC value. If it does not, the handler terminates.

Figure 8. PC-specific Part of the Software Stride Prefetch Handler

```
prefetch_address is loaded from the context of main thread;
i = min(prefetch count, calculated stride count);
do {
        prefetch_address = prefetch_address + stride distance;
        issue prefetch from prefetch_address;
} while (--i > 0);
return from trap;
```

The actions taken by the trap handler depends on the PC of the faulting instruction. The part of the trap handler code that is PC-specific is shown in Figure 8. The compiler generates code to compute the prefetch addresses based on the iteration count and the faulting address obtained from the main thread context.

We have simply ignored the synchronization problem between the main thread and the prefetching nanothread. It is therefore possible that some of the prefetches are fetching useless blocks, but this does not affect program correctness. The hardware simply ignores prefetches that cause protection traps. The number of prefetches is most of the time given by the default prefetch count, as in sequential prefetch, except towards the end of the loop where the number of prefetches depends on the number of iterations left.

4.3. Ideal prefetchers

The purpose of the simulations is to evaluate the effectiveness of nanotraps in emulating prefetching hardware. As a reference point, we also model ideal hardware prefetchers, which generate exactly the same prefetches as the software prefetcher in the second-level cache, but with no overhead. This can be achieved by executing the software handlers at the time of a cache miss outside of the simulator. Thus at the occurrence of a cache miss, the ideal prefetcher generates the addresses to prefetch and inserts them in a prefetch address queue in zero simulation time. The second-level cache then executes the prefetches one by one, inside the simulator.

5. Evaluation Methodology

We have run a detailed simulation model to experiment with assisted execution. The simulation environment is based on two separate simulators working together in a tightly-coupled fashion. A trace-driven simulator, called `superscalar`, implements a superscalar processor with support for assisted execution. It is driven by an execution-driven Sparc processor simulator, called `CacheMire-2` [1], which generates decoded instruction streams for main thread and nanothreads to `superscalar`. `Superscalar` sends requests for instructions to `CacheMire-2` with instruction count and thread identifier in the instruction fetch stage. Then, `CacheMire-2` executes the given thread by the given number of instructions and returns the decoded instructions to `superscalar`.

5.1. Benchmarks

We have run seven benchmarks from the SPEC95 benchmark suite [22]: two SPECint95 applications (`go`, `compress`) and five SPECfp95 applications (`swim`, `applu`, `su2cor`, `mgrid`, `wave5`). The code used for stride prefetching is generated by the Napai compiler developed at Halmstad University by Jonas Skeppstedt's group [15]. This compiler produces highly optimized code [20]. Each benchmark binary has a prologue and PC-specific trap handlers for stride prefetching, as well as a separate file for the hash table. The hash table contains 4096 entries but, in most cases, only about 10% of the entries are defined and conflicts are rare. The same benchmark binaries are used for the different prefetching strategies. For stride prefetching schemes, the content of the hash table is read into the data area, and the address of the prologue is

saved in NHAR (Figure 4) at the beginning of the simulation.

We simulate 100 million instructions of each benchmark (not counting nanothread code). The cache sizes are scaled differently for each benchmark. They are given in Table 2.

Table 2: Data Cache Sizes

	applu	compress	go	mgrid	su2cor	swim	wave5
FLC Size	4 KB	16 KB	16 KB	4 KB	2 KB	2 KB	4 KB
SLC Size	32 KB	64 KB	64 KB	32 KB	8 KB	8 KB	16 KB

For each benchmark and each memory latency, we report the results for the best prefetch count, which is shown in each figure.

5.2. Performance Metrics

The *execution time* is the main measure of performance. However, to gain better insight into the behavior of each benchmark, we also measure memory blocking time, degree of bad prefetch, nanotrap response time, nanothread activity, and thread concurrency.

The *memory blocking time* is the part of the execution time in which no instruction of the main thread makes progress in the pipelines. The *degree of bad prefetches* is the fraction of prefetched blocks that are never accessed before they are replaced or before the end of the simulation. *Nanotrap response time* is the average time between a primary cache miss and the sending of the first prefetch to the second level cache. *Nanothread activity* is the ratio between the number of nanothread instructions and the number of main thread instructions executed. *Thread concurrency* indicates the number of threads running concurrently. In every processor cycle, we record the number of threads active in the processor. At the end of the simulation, we compute the fraction of cycles that a given number of threads were active.

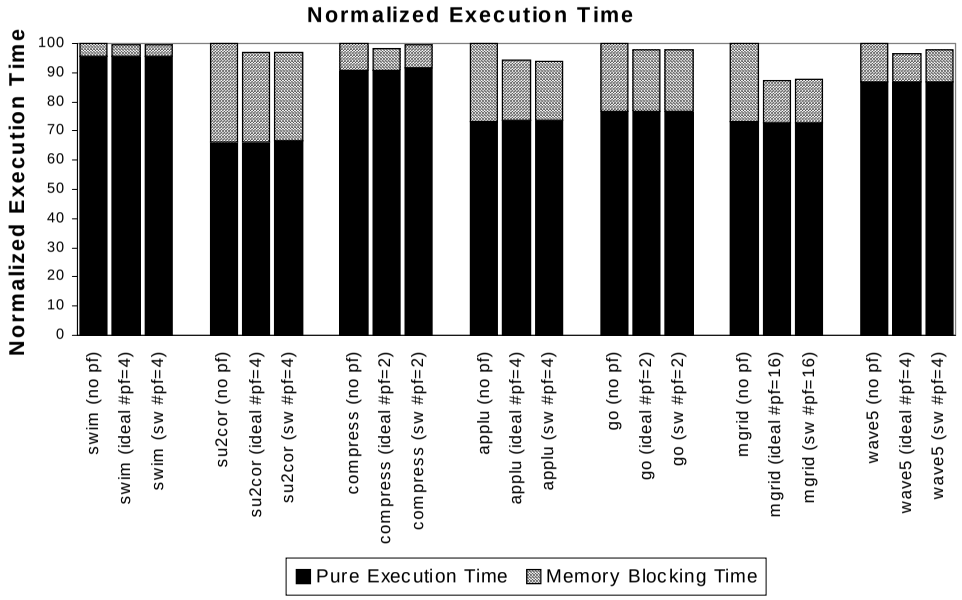
6. Simulation Results

6.1. Sequential Prefetch

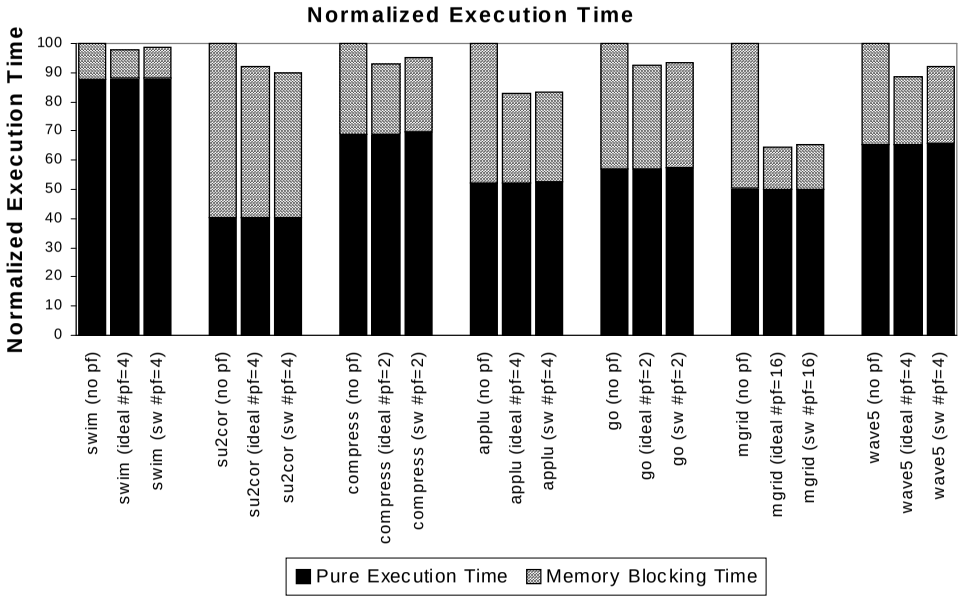
Figure 9 compares systems with software and ideal sequential prefetching to the same system with no prefetching. It appears that software sequential prefetch is as effective as the ideal hardware prefetchers. We first comment on the effectiveness of prefetching on the benchmark

execution times.

Figure 9. Normalized Execution Times and Instruction Dependencies (Sequential Prefetch)



(a) Memory Latency=50pclocks



(b) Memory Latency=200pclocks

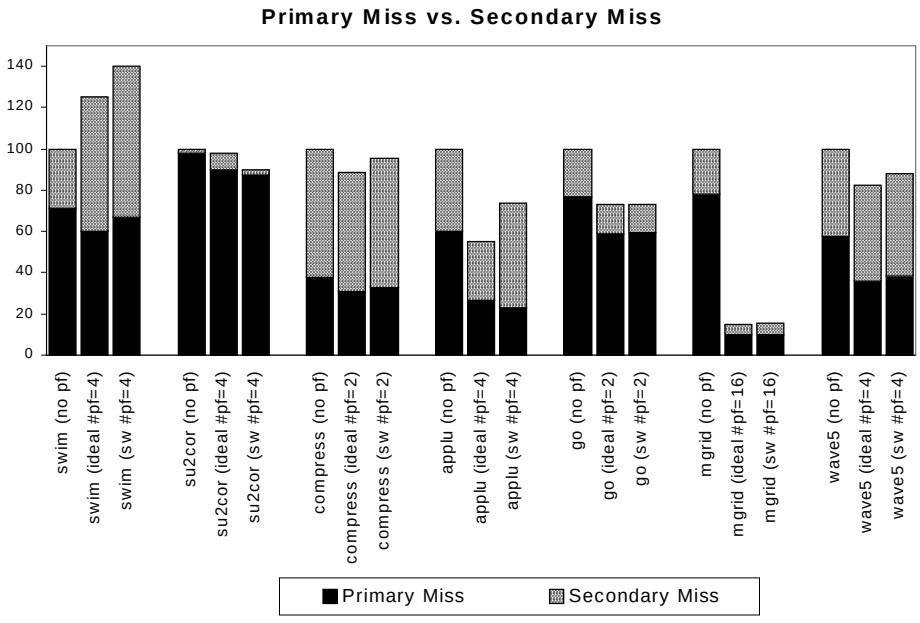
The effectiveness of sequential prefetching varies with applications. When an application exhibits very little memory blocking there is little to be gained by prefetching. This is clearly the

case for swim. Because swim's data miss rate is extremely low (see Table 3), its performance improves less than 2% with prefetching. Two benchmarks, applu and mgrid, show larger improvements due to prefetching. Even with a latency as low as 50 pclocks, the execution times improves by up to 13% in the simulations with prefetching. Other programs show moderate improvements. Similar trends were also observed and reported in [2].

The quality of the prefetches affects the performance directly. Table 3 and Table 4 show the miss rates and the degree of bad prefetches for the latency of 50 pclocks. The miss rates of ideal and software prefetching differ because the timing of the prefetches is different. For mgrid the miss rate with prefetching is cut down to less than 20% and the degree of bad prefetches is relatively low (54.63%) considering the prefetch count of 16. Even though sequential prefetching does not use application-specific stride information when deciding the blocks to prefetch, it is most effective for mgrid.

Sequential prefetching reduces execution time 1) by cutting the number of primary miss and 2) by turning some primary misses into secondary misses. As shown in Figure 10, in the case of wave5, the number of primary data misses is sharply reduced under prefetching.

Figure 10. Normalized Primary Miss and Secondary Miss Counts (Latency=50pclocks)



For swim and wave5, the number of secondary misses increases under prefetch while the

number of primary misses decreases. However, secondary misses do not affect performance much. Most of the time, the average penalty seen by secondary misses in prefetching cases is almost equal to the memory latency, which implies that most secondary cache misses tend to happen right after a primary cache miss on the same block. Accesses causing secondary misses must still wait in the address queue until the preceding primary miss retires and would have to wait even if they were hits. This explains why `Applu` with software prefetching experiences more secondary cache misses than with ideal prefetching but the execution time is not affected.

Table 3: Miss Rate (primary + secondary) (%) (Latency = 50 pclocks)

	swim	su2cor	compress	applu	go	mgrid	wave5
No Prefetch	0.28	3.29	3.66	3.55	2.13	4.29	2.70
Ideal Prefetch	0.36	3.21	3.25	1.96	1.56	0.65	2.22
SW Prefetch	0.40	2.97	3.50	2.61	1.56	0.66	2.38

Table 4: Degree of Bad Prefetches (%) (Latency = 50 pclocks)

	swim	su2cor	compress	applu	go	mgrid	wave5
Bad Prefetch	75.87	79.60	79.16	40.87	57.60	54.63	79.04

These observations agree with previously published results in different environments. It is known, for example, that the execution time of ILP processors is not that sensitive to the penalty experienced by individual memory access instructions [19]. Similarly, in [16] software-controlled prefetching [14] is shown to be less effective in reducing the memory blocking component of execution time than in traditional processor.

6.2. Nanothread Activity

The nanothread activity is displayed in Table 5. This activity is quite low and we believe that nanothreads could efficiently implement much more complex mechanisms than sequential prefetching. Nanothreads work harder in favor of `su2cor` because most cache misses in `su2cor` are primary misses (Figure 10) and its miss rate is high as shown in Table 3.

Table 5: Nanothread Activity (%)

	swim	su2cor	compress	applu	go	mgrid	wave5
Activity	0.67	9.11	2.11	3.87	1.98	5.99	3.96

Table 6: Thread Concurrency (%)

	swim	su2cor	compress	applu	go	mgrid	wave5
1 thread	95.59573	79.30909	92.72737	89.96152	91.65257	92.77511	88.23303
2 threads	3.09828	16.17952	3.65240	7.49005	7.96112	3.60884	5.34133
3 threads	0.56539	4.42414	2.77154	2.52565	0.29383	2.85355	4.92720
4 threads	0.66068	0.07493	0.60405	0.02088	0.04541	0.39372	1.35182
5 threads	0.07615	0.00905	0.24452	0.00119	0.02178	0.16088	0.12832
6 threads	0.00199	0.00243	0.00004	0.00008	0.01537	0.05362	0.01186
7 threads	0.00112	0.00039	0.00004	0.00027	0.00921	0.06212	0.00377
8 threads	0.00066	0.00045	0.00005	0.00036	0.00071	0.09215	0.00266

Nanothreads compete with the main thread in the instruction fetch scheduler and the instruction issue scheduler, and cause structural hazards. However Figure 9 comparing the execution times of nanothread-based prefetching and ideal prefetching shows that these conflicts do not slow down the main thread significantly, even in the case of `su2cor`. The reason is that most of the time the main thread runs alone or concurrently with a few nanothreads. Table 6 shows the thread concurrency. *i thread* means that the main thread plus (*i* -1) nanothreads are active. We observe that the time during which less than three nanothreads are working with the main thread covers 99% of the total execution time, and that the main thread runs alone more than 80% of the time. Four nano-PC would have been sufficient to support sequential prefetching in this architecture.

Table 7: Latency in pclocks until the First Prefetch (Memory Latency=50 pclocks).

	swim	su2cor	compress	applu	go	mgrid	wave5
Ideal Prefetch	4.00	4.03	4.04	4.34	4.06	7.23	4.83
SW Prefetch	45.51	35.88	21.55	56.93	37.02	47.51	52.73

Table 7 shows the nanotrap response time. The latency between the primary miss and the first prefetch request in the ideal case is not affected directly by processor activity because the prefetches are triggered and executed in the cache controller. The only delay in this case is the queueing time in the prefetch queue and possible backup of PMAQ.

By contrast, prefetching nanothreads must first get through the instruction queues. If any instruction queue is full at the occurrence of a nanotrap, the instruction fetch stage stalls and the

nanothread is also delayed. Hence, longer memory access latencies further delay the response time of nanothreads. This delay may have some effects on the execution times

6.3. Stride Prefetch

The speedup due to prefetching is dependent upon the efficiency of the prefetching algorithm. Table 8 shows the comparison between four different prefetch configurations: no prefetching, ideal stride prefetching, software stride prefetching and software sequential prefetching for the case of `applu`. When applicable, the numbers in parenthesis are relative to the corresponding value for the system with no prefetch, except for the memory blocking time, which is relative to the execution time of the system with no prefetch.

The nanothread response times are 194.4 and 237.9 pclocks for sequential and stride prefetching respectively. The main reason why the response times are so long is that, most of the time, instruction queues are saturated, especially the address queue. In `applu`, the average address queue occupancy is about 11 entries out of the queue size of 12 entries throughout the execution time. This implies that the address queue is very likely to be full when a cache miss occurs, and the processor stalls. In this case, nanothread instructions can be fetched only after the missing data returns from memory and the instruction retires to make room for other instructions. The stride prefetching handler executes much more instructions than the sequential prefetch handler, which explains the difference in prefetch latencies of these two approaches.

Table 8: Comparison of Stride and Sequential Prefetch (for `applu`, Latency=200 pclocks)

`applu`	No Prefetch	Ideal Stride	Software Stride	Software Sequential
Execution time	145335277 (100.00%)	119257176 (82.06%)	119951074 (82.53%)	121273373 (83.44%)
Memory blocking time	69345696 (47.71%)	43201454 (29.73%)	43870084 (30.19%)	45068436 (31.01%)
Miss rate	3.55 (100.00%)	2.19 (61.69%)	2.21 (64.51%)	2.68 (75.49%)
Primary miss rate	2.14 (100.00%)	0.93 (43.46%)	1.04 (48.60%)	0.82 (38.32%)
Degree of bad prefetch (%)	(NA)	18.3	19.3	40.3
Issue rate	0.688 (100.00%)	0.839 (121.95%)	0.834 (121.22%)	0.825 (119.91%)
Instruction dependency	7.559 (100.00%)	6.827 (90.32%)	6.739 (89,15%)	6.819 (90.21%)
Nanothread activity. (%)	(NA)	(NA)	7.941	3.873
Nanothread response time	(NA)	(NA)	237.9	194.4

Among all the benchmarks `applu` is the one for which stride prefetching works best. As

indicated by the miss rate and the degree of bad prefetches, stride prefetching is more accurate than sequential prefetching. Although the stride prefetching handler executes more instructions than the sequential prefetching handler, this added work is mostly hidden by the highly concurrent execution of instructions in the processor [16].

7. Related Work

The work previously published and relevant to this paper falls in three categories: memory informing operations, prefetching and simultaneous multithreading.

Horowitz, *et al*. [8] proposed *memory informing operations* to help software observe the memory referencing behavior by trapping on selected first-level cache misses and by taking actions when needed in the trap handler. Through simulations, they showed that the overhead of the memory informing trap handler is less than 40% of the total execution. One of the major reasons that memory informing operations are penalized by this overhead is that the processor model supports only one single thread at a time. Moreover, traps are expensive. In some sense we could say that nanotraps implementing sequential and stride prefetching are extensions of the memory informing operations to ILP processors with simultaneous multithreading. However, nanotraps and nanothreads are more general since they are not restricted to deal with events caused by the memory system.

Skeppstedt and Dubois [20] exploited the idea of memory informing operations in a traditional pipelined processor executing a single thread at a time to propose a hybrid software/hardware stride prefetching scheme in multiprocessors. Second-level cache misses trigger traps which either program and start an autonomous hardware stride prefetcher or issue the stride prefetches. The trap handlers run while the processor waits on memory access misses in a sequentially consistent system, which is not possible in an ILP processor. In this paper we have adopted the same stride prefetching algorithm in nanothreads. With assisted execution, the scheme becomes feasible and efficient in ILP processors.

Dahlgren and Stenstrom [4] compared sequential and stride prefetching for shared-memory multiprocessors. They showed that sequential prefetching outperforms stride prefetching for many applications, because most strides are shorter than the block size if the cache block is large enough and because sequential prefetching can exploit the locality of misses with non-stride

accesses. Because sequential prefetching raises the traffic in the memory system stride prefetching may be superior to sequential prefetching under limited bandwidth. In a uniprocessor environment, where memory bandwidth issues are different, we have also observed that stride prefetching is more accurate but is not always as good as sequential prefetch. In the best case for stride prefetching, the case of `applu`, the software sequential prefetching is still very competitive, as shown in Table 8. However we did not simulate memory conflicts.

In [2], Charney and Puzak show instruction and data cache miss rates for the SPEC 95 benchmark suite on a traditional processor and evaluate two prefetching algorithms: next-sequential prefetching (NSP) and shadow-directory prefetching (SDP). The former approach is very similar to our sequential prefetch algorithm. Since they skip the first billion instructions in the program and simulated the next 500 million instructions, a direct comparison of their results with ours is difficult. However, some observations are consistent with ours. They show that prefetching is less effective for `compress` and `go` because these applications lack a next-sequential miss pattern. We also observed that the performance of these programs is worse as the prefetch count increases. The degree of bad prefetch of `compress` is the highest among all the benchmarks even with a small prefetch count. On the other hand, `mgrid` and `applu` do very well under the NSP scheme as well as in nanothread based sequential prefetch.

Ranganathan, *et al*. [16] show that software-controlled non-binding prefetching can be very effective for some applications in ILP processors. The overhead of prefetch instructions is less than in traditional processors because the execution of prefetch instructions is overlapped with other computation or memory accesses. Instead of adding prefetch instructions in the programs, we rely on nanotraps, which generate prefetches on cache misses only, tracking the dynamic application behavior. The effect of data misses in out-of-order superscalar processors was also studied by André Seznec and Fabien Lloansi [19].

The original goal of multithreaded processors was to eliminate processor blocking time due to program dependencies, which leads to a vast waste of hardware resources. Simultaneous multithreaded (SMT) processors were introduced by Tullsen et al. in [23]. SMT relies on the ability of parallelizing compiler to produce enough threads to exploit the processor resources and on the degree of parallelism embedded in application programs [5] [24] [6]. Pedro Marcuello and Antonio Gonzalez’s multithreaded processor exploits the parallelism found in highly predictable

branches such as loops without relying on special instructions or parallelizing compilers [12]. The assisted execution model is based on the general SMT model in that multiple threads runs concurrently by sharing the given processor resources but the parallelism comes from work added to the computation.

8. Conclusions and Future Research

Assisted execution is a new execution paradigm. Assistant threads called nanothreads are attached to application threads. In the context of ILP processors and simultaneous multithreading, these nanothreads can boost the execution speed of the main thread while their instruction overhead is hidden by the concurrent execution of instructions in the processor. Assisted execution generalizes previous proposals such as memory informing operations and hybrid software/hardware prefetching strategies and provides a framework in which they can be more efficient.

Static, compile-time execution mechanisms are often preferred because dynamic mechanisms require additional work, which can easily offset their potential gains. We believe that assisted execution offers brand new opportunities for dynamic mechanisms. In a multiprocessor environment, nanotraps can be used to execute coherence protocol handlers [10]. By monitoring and predicting data and instruction access patterns based on compile-time knowledge nanothread may help reduce the cost of conditional branches and of data access penalties in irregular applications. Systems with adaptive hardware such as adaptive protocols, variable cache block size, or configurable interconnects may become feasible, since the overhead of monitoring and configuring is hidden in nanothreads. Dynamic execution profiling and software reconfiguration can also be done efficiently with nanothreads; in this approach called adaptive execution in [18], profiling information is gathered dynamically and is fedback to the main thread to affect its execution by selecting a different code segment. Finally, nanothreads could be used for other purposes than enhancing performance, such as low-overhead online diagnosis, and fault-tolerance by low-overhead online fault detection and hardware reconfiguration.

For the nanothread algorithms evaluated in this paper, synchronization was not a problem. In general however efficient synchronization mechanisms between nanothreads and main thread must be designed and evaluated, in the light of more complex dynamic execution mechanisms.

Another problem not addressed in this paper is the programming problem. Namely we

need to create a programming environment in which nanothreads can be developed and attached to main threads. Whether a compiler can do this automatically is of course the main challenge.

Acknowledgment

This work was funded by the National Science Foundation under Grant No. MIP-9633542. We want to thank Mary Hall of the Information Science Institute and Rafael Saavedra from the Computer Science Department at U.S.C. for many lively discussions on the topic of assisted execution and for helping in shaping the main ideas in this paper. Also, Jonas Skeppstedt from Halmstad University (Sweden) provided us with compiled code without which this work would have been impossible.

9. References

[1] M. Brorsson, F. Dahlgren, H. Nilsson, and P. Stenström, "The CacheMire Test Bench - A Flexible and Effective Approach for Simulation of Multiprocessors," *Proceedings of 26th Annual Simulation Symposium*, pp. 41-49, March 1993.

[2] M. J. Charney and T. R. Puzak, "Prefetching and memory system behavior of the SPEC95 benchmark suite," *Performance analysis and its impact on design*, Vol. 41, No. 3, 1997

[3] T-F Chen and J-L Baer, "A Performance Study of Software and Hardware Prefetching Schemes," *Proceedings of the 21th International Symposium on Computer Architecture*, pp. 223-232, May 1994.

[4] F. Dahlgren and P. Stenström, "Evaluation of Hardware-Based Stride and Sequential Prefetching in Shared-Memory Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 7, No. 4, pp. 385-398, April 1996.

[5] S. J. Eggers, et al., "Simultaneous Multithreading: A Platform for Next-generation Processors," *IEEE Micro*, pp. 12-18, September/October 1997.

[6] B. Goossens, "Tipi: The Threads Processor," *MTEAC '98 Conference*, 1998.

[7] L. Gwenlapp, "Dansoft Develops VLIW Design," *Microprocessor Report*, Vol. 11, No. 2, Feb. 17, 1997, pp. 18-22.

[8] M. Horowitz, M. Martonosi, T. C. Mowry, and M. D. Smith, "Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors," *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pp. 260-270, May 1996.

[9] D. Kroft, "Lockup-free Instruction Fetch/Prefetch Cache Organization," *Proc. of the 8th Int. Symposium on Computer Architecture, pp. 81-87, May 1981*.

[10] J. Kuskin et al. The Standford FLASH Multiprocessor. *Proc. of the 21st Annual International Symposium on Computer Architecture*, pages 302-313, April 1994.

[11] C. Luk and T.C. Mowry, "Compiler-Based Prefetching for Recursive Data Structures," Proceedings of ASPLOS'96, Oct. 1996.

[12] P. Marcuello and A. González, “Control and Data Dependence Speculation in Multithreaded Processors,” *MTEAC ‘98 Conference*, 1998.

[13] MIPS Technologies Inc., “R10000 Microprocessor User’s Manual-Version 2.0,” December 1996.

[14] T. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, Computer Systems Laboratory, Stanford, CA, March 1996.

[15] “The Napai Compiler Project” Halmstad University, Sweden, http://www.hh.se/staff/jonas/napai/index.html.

[16] P. Ranganathan, V. S. Pai, H. Abdel-Shafi, and S. V. Adve. “The Interaction of Software Prefetching with ILP Processors in Shared-Memory System,” *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[17] S.K. Reinhardt, J.R. Larus, and D.A. Wood. Tempest and Typhoon: User-Level Shared Memory. *Proc. of the 21st Annual International Symposium on Computer Architecture*, pages 325-337, April 1994.

[18] R.H. Saavedra and D. Park, “Improving the Effectiveness of Software Prefetching with Adaptive Execution, “ *1996 Parallel Architecture and Compilation Techniques (PACT’96)*, Oct. 1996.

[19] A. Seznec and F. Lloansi, “About Effective Cache Miss Penalty on Out-Of-Order Superscalar Processors,” *TR IRISA-970*, November 1995.

[20] J. Skeppstedt and M. Dubois, “Hybrid Compiler/Hardware Prefetching for Multiprocessors Using Low-Overhead Cache Miss Traps,” *Proc. of the 1997 Int. Conf. on Parallel Processing*, pp.298-305.

[21] P. Song, “Multithreading Comes of Age,” *Microprocessor Report*, Vol. 11, No. 9, Jul. 14, 1997, pp. 13-18.

[22] The SPEC Corporation, *The SPEC95 Benchmark Suite*, 1995.

[23] D. M. Tullsen, S. J. Eggers, and H. M. Levy, “Simultaneous Multithreading: Maximizing On-Chip Parallelism,” *Proceedings of the 22rd Annual International Symposium on Computer Architecture*, pp. 392-403, June 1995.

[24] D. M. Tullsen, et al., “Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor,” *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pp.191-202, May 1996.