DECEMBER 1994

WRL
Research Report 94/8

How Useful Are Non-blocking Loads, Stream Buffers, and Speculative Execution in Multiple Issue Processors?

Keith I. Farkas
Norman P. Jouppi
Paul Chow

digital **Western Research Laboratory** 250 University Avenue Palo Alto, California 94301 USA

How Useful Are Non-blocking Loads, Stream Buffers, and Speculative Execution in Multiple Issue Processors?

Keith I. Farkas*
Norman P. Jouppi
Paul Chow*

December 1994

*Keith I. Farkas and Paul Chow are with the Dept. of Electrical and Computer Engineering, University of Toronto, 10 Kings College Road, Toronto, Ontario, Canada, M5S 1A4.

Western Research Laboratory 250 University Avenue Palo Alto, California 94301 USA

Abstract

We investigate the relative performance impact of non-blocking loads, stream buffers, and speculative execution both used individually and in conjunction with each other. We have simulated the SPEC92 benchmarks on a statically scheduled quad-issue processor model, running code from the Multiflow compiler. Non-blocking loads and stream buffers both provide a significant performance advantage, and their combination performs significantly better than either alone. For example, with a 64-byte, 2-way set associative cache with 32 cycle fetch latency, non-blocking loads reduce the run-time by 21% while stream-buffers reduce it by 26%, and the combined use of the two yields a 47% reduction. The addition of speculative execution further improves the performance of the systems that we have simulated, with or without non-blocking loads and stream buffers, by an additional 20% to 40%. We expect that the use of all three of these techniques will be important in future generations of microprocessors.

1 Introduction

A continuing trend in the design of computer systems is the widening gap between microprocessor and memory speeds. This speed discrepancy can have a significant impact on the performance of a system since it increases the time cost of servicing data cache misses. The performance impact can be lessened through the use of techniques that reduce the amount of time the processor is stalled for cache misses. In this paper, we consider non-blocking loads, stream buffers, and speculative loads. We investigate the relative merits of these three techniques when used individually and in conjunction with each other, in the context of an advanced quad-issue statically scheduled microprocessor.

Non-blocking loads reduce the time stalled due to cache misses by allowing the processor to overlap the servicing of a miss with the execution of other instructions. The amount of overlap depends on the number of instructions that are available for execution that do not use the register being targeted by the load instruction. When an instruction is encountered that depends on the value being loaded and the load is still in-progress, the processor must stall until the load completes; a *true-data dependency* is said to exist between these two instructions. With a lockup cache, a stall will also occur if during the processing of a cache miss, other load or store instructions are executed. Such stalls can be avoided by using a *lockup-free cache*. A lockup-free cache allows multiple concurrent cache hits, misses, or both. By overlapping the processing of cache misses, the average time to service a miss decreases because the processor will be stalled for less time and more loads will complete in this time.

Prefetching also can reduce the time stalled due to cache misses. The goal of prefetching is to bring data closer to the processor before the processor requires it, rather than, as is the case with non-blocking loads, waiting for a cache miss to occur before initiating a fetch for missing data. The prefetch is initiated by some triggering event. With software prefetching, the trigger is prefetch instructions that are inserted into the code by a sophisticated compiler or user [1, 2]. With hardware prefetching, hardware is used to determine when a prefetch might be useful. Software prefetching has been most successful on numeric codes, while hardware prefetching can be used with all types of applications (including the operating system).

Examples of hardware prefetch techniques include Chen and Baer's lookahead PC reference prediction method [3] and stream buffers [4, 5]. We study stream buffers as we believe them to be simpler and less-invasive than the lookahead scheme. The lookahead scheme is complicated by the need for additional ports into the data-cache tags when used in a superscalar processor. In such a processor, the cache-tag ports are one of the most critical resources, and increasing their number may result in a physically larger cache and/or a slower cache, or may be infeasible. Stream buffers, on the other hand, sit on the memory side of the data cache and while they must be probed on every data reference, they do not need access to the cache tags. Stream buffers trigger a prefetch based on previous cache misses. Also, they avoid polluting the cache by placing prefetched data in special buffers.

Unlike non-blocking loads and stream buffers, speculative execution is a software-based technique. It involves moving code beyond branches (see [6] for more details). Speculative execution of load instructions is not the same as software prefetching using non-blocking loads; the former involves the movement of *existing* load instructions while the latter involves the insertion of *ad-*

```
L2   subi   r1, 1, r1      ; r1 ← r1-1
     bnz    r1, L1         ; branch if r1 is non zero   (1)
     ld     r6, 700(sp)    ; r6 ← [sp+700]               (2)
     ld     r8, 732(sp)    ; r8 ← [sp+732]               (3)
     addi   sp, 8, sp      ; sp ← sp+8                   (4)
     mult   r8, r8, r4     ; r8 ← r8 × r4                (5)
     br     L2             ; branch always
```

Figure 1: Code segment for a RISC single-issue processor.

ditional load instructions. Speculative execution can increase the performance of a program in a number of ways. In superscalar designs, the instruction-level parallelism can be increased if there is a sufficient amount of hardware parallelism available. This increase can also provide more flexibility when scheduling load instructions for a machine with extensive support for non-blocking loads. The result of this flexibility is a better tolerance of the cache-miss latency and a reduction in the average per-miss penalty due to allowing more misses to be simultaneously outstanding.

The three techniques we consider can be used alone or together. To illustrate their use, consider the code segment in Figure 1 which, for simplicity, corresponds to a single-issue machine. Assume that both loads miss in the cache and that the miss penalty is 10 cycles. With none of the three techniques in use, the processor has to stall twice and each time for the full length of the miss penalty. If, however, non-blocking loads are used with a lockup-free cache, the processor will have to stall only once, that is, when it tries to execute instruction 5 (due to a true-data dependency between instructions 5 and 3). If instead stream buffers are used, there will still be two stalls but the cache miss caused by the first load will initiate the prefetch for the data required by the next load (see Section 2.2 for details on how a stream buffer works). Hence, the time to service the second load is reduced. Finally, if both non-blocking loads and stream buffers are used together, there will be only one stall and the stall time will be smaller than it was with only non-blocking loads. Stream buffers used alone or together with non-blocking loads also reduce the stall time for subsequent iterations of the loop because the prefetch for the required data will have been initiated in a previous iteration.

Speculative execution is useful for increasing the instruction-level parallelism and for improving the effectiveness of non-blocking loads. The first of these effects cannot be applied to this example because we are assuming a single-issue machine. To show the second one, assume we are using non-blocking loads. With speculative execution, the compiler can move the two load instructions above the branch and thus execute them earlier. The end result is that the processor will stall for less time when it executes instruction 5. The use of stream buffers reduces the stall time further because a prefetch of the data for instruction 3 will be issued before this load is executed.

Previous Work

Other researchers have investigated non-blocking loads, prefetching, or speculative loads, but not their combination. Rogers and Li [7] investigated software support for speculative loads with non-blocking caches using many of the Livermore loop kernels; they compared the results to blocking caches. Sohi and Franklin, on the other hand, studied non-blocking loads [8], while

Callahan and Mowry have studied software prefetching for scientific codes [1, 2]. Chen and Baer [3] investigated a combination of non-blocking loads and prefetching, but used a lookahead-PC reference prediction method with a production compiler, instrumented with Pixie, and rescheduled only at a basic block level. Comparing the effectiveness of the different techniques used in these studies is difficult because of different assumptions used by each group of researchers, and the fact that no group looked at all three techniques.

In this paper we look at the relative memory-system performance improvement available from non-blocking loads, hardware prefetching, and speculative execution used individually and in combination. We do this investigation in the context of an advanced quad-issue superscalar machine, and a pipelined memory system made with recent RAM techniques such as synchronous DRAMs. An important part of this study was the advanced Multiflow Compiler technology [9], which provided trace scheduling and support for speculative loads.

2 Simulation Methodology

We investigated the relative performance of non-blocking loads and stream buffers by examining how their use would affect the performance of current state-of-the art microprocessor systems. This investigation was carried out by simulating a number of machine configurations using a processor model that resembles a number of commercial processors including the PowerPC 604 [10], the DEC EV5 [11], the MIPS T5 [12] and the SUN Ultrasparc [13]. All configurations used the same hardware for servicing instruction requests and executing instructions. As a result, the contribution to the execution time of a benchmark from instructions was constant for all configurations. The machine configurations differ in the hardware that is available for servicing the processor's requests for data. This fact allows us to better estimate the impact of stream buffers and non-blocking loads.

The processor model implements a RISC processor that can issue four instructions per cycle and uses a conventional, statically scheduled, pipeline. Static scheduling is assumed since the performance benefits of dynamic scheduling are not sufficiently clear in view of the impact the more complex pipeline will have on the cycle time of the processor. For example, the statically scheduled DEC Alpha 21064A achieves a clock frequency of 275MHz in a 0.5um technology, while the dynamically scheduled PowerPC 604 achieves a clock frequency of 100MHz in a 0.5um technology. (There are many other differences between these machines, but in general dynamically scheduled machines announced to date have had significantly slower cycle times than statically scheduled machines in the same technology generation.)

The processor we model supports non-blocking stores and can be configured to support non-blocking loads. There are separate instruction and data caches with the instruction cache having a fixed miss penalty. The data cache is always lockup-free irrespective of whether the processor uses blocking or non-blocking loads[1].

The model also can include one or more stream buffers and these are used to prefetch *data*. The stream buffers may use either unit strides or dynamically calculated strides. Finally, data is fetched

[1]The only difference between these two modes of operation is that with blocking loads, no subsequent instruction can execute until the load is resolved. This restriction allows us to use the same cache model for both modes.

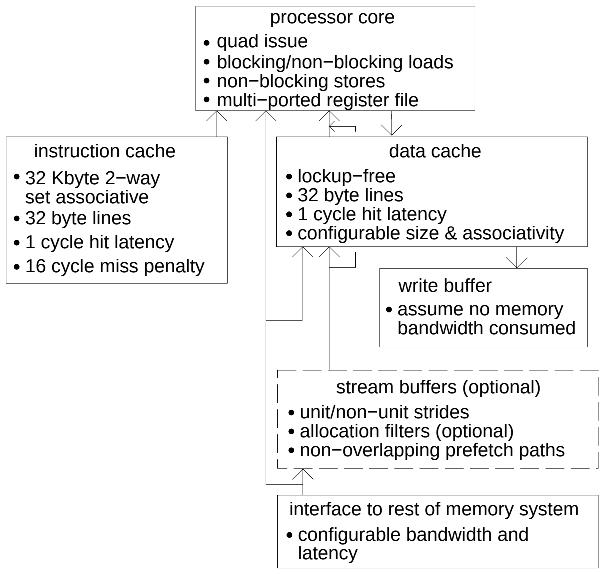

processor core
• quad issue
• blocking/non−blocking loads
• non−blocking stores
• multi−ported register file
instruction cache
• 32 Kbyte 2−way set associative
• 32 byte lines
• 1 cycle hit latency
• 16 cycle miss penalty
data cache
• lockup−free
• 32 byte lines
• 1 cycle hit latency
• configurable size & associativity
write buffer
• assume no memory bandwidth consumed
stream buffers (optional)
• unit/non−unit strides
• allocation filters (optional)
• non−overlapping prefetch paths
interface to rest of memory system
• configurable bandwidth and latency

Figure 2: Overview of machine model.

from the next lower level in the memory hierarchy through a bandwidth limited interface. Figure 2 presents an overview of the above machine model. Some of the model details are described further below.

2.1 Processor and Memory Models

Of the four instructions issued per cycle by the processor, each instruction word can contain at most: four integer operations, one floating-point division operation, two floating-point operations, two memory operations (i.e., two loads, two stores, or one of each), and one control flow operation (i.e., branch, subroutine call or return). All integer functional units have single-cycle latencies except for the multiply unit, which is fully pipelined and has a six-cycle latency. All floating point units have three-cycle latencies and are also fully pipelined, with the exception of the floating-point divider. The floating-point divider is not pipelined and has an eight-cycle latency for 32-bit divides, and a 16-cycle latency for 64-bit divides. Finally, stores take one cycle to be resolved and there is a single load-delay slot.

The instruction cache is 32K-byte, 2-way set associative with a 1-cycle cache-hit latency and a 16-cycle cache-miss latency. In addition to cache-miss induced stalls, branch instructions may also introduce stalls if the branch-delay slot(s) cannot be filled by the compiler or if the branch direction is mispredicted. In view of the high correct prediction rates reported by McFarling [14] and Yeh and Patt [15], we assume that all branches are correctly predicted dynamically with the exception of 5% of conditional branches. For conditional branches, the model associates a two cycle stall with each mispredict. We implement this penalty during the simulation of a benchmark by adding a single-cycle stall for every 10^{th} conditional branch that is executed.

Stores are assumed to be implemented using write-around (i.e., no-write-allocate) and write-through policies with a write buffer situated between the data cache and lower levels in the memory hierarchy. Since our goal is to compare the effectiveness of stream buffers and non-blocking loads while keeping constant other contributions to the execution time, we assume that no memory bandwidth is required to retire stores in the write buffer. This assumption prevents any stalls due to a full write buffer and prevents stores from delaying the servicing of stream buffer or cache fetches.

The lockup-free data cache can resolve cache hits in a single cycle and employs an inverted MSHR (Miss Status Holding Register) organization [16] to process cache misses. An inverted MSHR organization can support as many in-flight cache misses as there are registers and other destinations for data in the processor. Hence, there can be a cache miss outstanding to each of the processor's 32 integer and 32 floating-point registers. The register file has eight read ports and sufficient write ports to prevent any write-port conflicts arising when registers are filled on the resolution of a cache miss.

Requests for blocks of data are sent via the memory interface to the next level in the memory hierarchy. The memory interface returns the requested block in a constant number of cycles, called the *fetch latency*. The bandwidth of the interface is constrained by controlling the number of cycles between the launching of fetch requests. A *fetch spacing* of one allows the memory interface pipeline to be full whereas a spacing equal to the fetch latency allows at most one in-flight fetch. Thus, the time required to resolve a cache miss is not deterministic but has a lower bound equal to the fetch latency. When a block is returned to the cache, the cache line is written simultaneous with the writing of the appropriate words into all registers with loads outstanding to this block (updating all pending registers requires the multiple write ports mentioned above). This simultaneous writing is represented in Figure 2 by the arrows that bypass the data cache. Writing a register or a cache line is assumed to take one cycle.

2.2 Stream Buffers

Our stream buffer model is based on the model originally proposed by Jouppi [4] with the four enhancements described below; Appendix A describes our stream buffer implementation in detail. In the original model, stream buffers consist of a number of entries that are managed as a FIFO queue. Each entry in the queue can store a block of data and the corresponding address. All entries at the head of the stream-buffer queues are probed at the same time as the data cache probe is done. If the data cache probe results in a hit, the stream buffers are not touched. However, if a data cache miss occurs, and the desired block is in a head entry, the cache block is read out and it is written into the cache. The stream buffer then issues a prefetch request to the next lower level in the memory hierarchy to fill the empty entry with subsequent blocks.

When a miss occurs to both the cache and the stream buffers, a request for the block containing the miss address is issued. Then, a stream buffer is allocated and told to begin fetching blocks subsequent to the missing block. Because each block that is fetched has a block-address one greater than the last, the stream buffers are said to use a *unit stride*. Once the request for the first block to be prefetched is launched into the memory subsystem, the stream buffer can issue another prefetch request if there remain empty entries in the queue.

Enhancements to the Original Model

In this study we have made four enhancements to the original stream buffer model:

1. allocation filters
2. hardware support for dynamic strides
3. the enforcement of non-overlapping prefetch paths
4. direct access to the stream buffer entries.

These enhancements are described below.

First, in the original stream buffer proposal, a stream buffer is allocated whenever a data reference misses in the cache and in the entries at the head of the stream buffer queues. This allocation policy can result in prefetching down a subsequently unused stream should the data reference be isolated. Prefetching down such a stream will generate excess traffic to the memory system as well as potentially discarding useful data from a previously allocated stream buffer. To prevent this situation from occurring, an *allocation filter* can be used. We implement the filter proposed by Palacharla and Kessler [5]. This filter prevents a stream buffer from being allocated until two misses occur for the same stream. On the second miss, a stream buffer is allocated and it begins prefetching the block subsequent to the one corresponding to the second miss.

Second, instead of limiting prefetch strides to the unit stride of the original proposal, the stride can be determined dynamically based on previous miss addresses. We implemented a scheme based based on the *minimum delta* scheme proposed by Palacharla and Kessler [5]. With this scheme, on a stream buffer miss, the allocation filter is applied to determine whether a unit-stride should be used. If there is a filter miss, then the minimum signed difference between the miss address and the last N miss addresses is determined; this minimum delta, which may be positive or negative, is the stride. In our model, a stream buffer is allocated if the miss is the third miss in a series to blocks that are separated by this stride. Our stream buffer model with the filter and stride predictor is shown in Figure 3.

Third, when there are multiple stream buffers, we ensure that a given block of data resides in at most one stream buffer. In other words, the stream buffers always prefetch down non-overlapping paths. Non-overlapping paths prevent duplication and thus ensure that the maximum benefit is obtained from the available stream buffers. To achieve non-overlapping paths, a comparator must be associated with each stream buffer entry. While these comparators increase the design complexity, in practical systems, they need to be included for enforcement of multiprocessor cache consistency anyway.

Fourth, we have extended the original stream buffer proposal to include direct access to non-head entries in the queues (this extension is not shown in the figure). This extension reduces the time required to load the cache with data not in the entry at the head of the queue since there is no need to first shift out the blocks closer to the head. We assume that it takes one cycle to extract a block of data from a stream buffer.

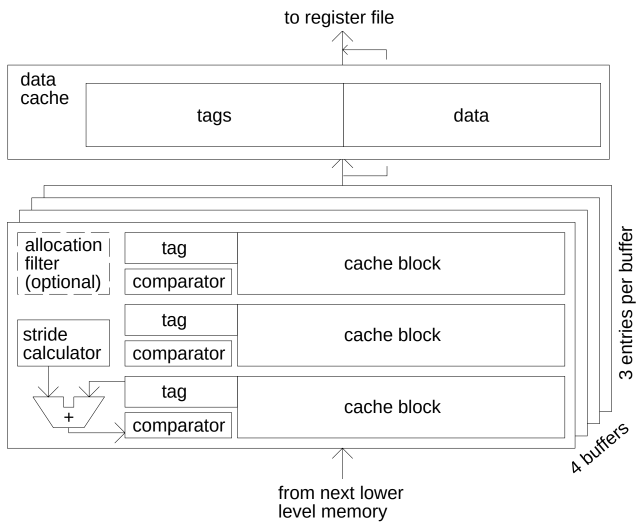


Figure 3: The stream buffer model: an example with 4 stream buffers each with 3 entries.

2.3 Simulation Framework

To perform the simulations for this study, we used an *object-code translation and instrumentation system*. This system emulates the execution of a benchmark as it would run on a target machine by running the benchmark on an existing machine. As a result *both* the functional behavior and the memory behavior of the application are simulated. The first step in performing a simulation is to compile the benchmark using instruction scheduling rules pertaining to the architecture of the processor to be modeled. We use a modified version of the Multiflow VLIW Compiler [9] for this purpose[2]. Next, the resulting assembly language (i.e., object code) is translated into the assembly language of the machine on which the simulations are run, namely, Alpha AXP workstations. Instrumentation and modeling code is then inserted into the translated code. Finally, the augmented, translated binary is linked with similarly compiled and instrumented run-time libraries and support routines.

The instrumentation code is inserted to record the emulated run-time behavior of the benchmark. This code records various statistics including cache miss rates, the number of (simulated) instructions executed, and the number of (simulated) clock cycles. The modeling code is inserted to allow the factoring in of the time required to resolve memory and register accesses. This modeling is accomplished by inserting before every emulated load and store instruction a call to a procedure that models the memory. These calls pass to the procedure the address of the item being loaded or stored and the procedure returns the amount of time required to process the access. For example, for non-blocking loads, this time will be the time required to launch the load whereas for a blocking-load it will be the time required to load the data into the cache if it is missing. A mechanism in the simulator adjusts these addresses so that they do not reflect the presence of the simulation infrastructure. Calls to a scoreboard procedure are also inserted before every emulated instruction that uses the result of a load. This procedure factors in the time required to validate the source registers of the instruction.

[2]The compiler was modified to produce RISC-like object code for a processor with 32-bit addresses, 32-bit integers and 64-bit floating-point numbers. The compiler uses a common backend for both C and Fortran code.

Abbrev.	Processor Details	
	blocking loads	stream buffers
un	yes	none
sb		unit stride
sb+fds		filter & dynamic stride
nbl	no	none
nbl+sb		unit stride
nbl+sb+fds		filter & dynamic stride

(a) Processor Designs

Abbrev.	Memory System Details		
	cache	fetch	
		spacing	latency
ideal	assume 100% data cache hit rate		
8DM	8 KB direct mapped	1	8
		8	32
64SA	64 KB 2-way associative	1	8
		8	32

(b) Memory Configurations

Table 1: System details with associated abbreviations.

3 Performance Trends

We investigated four processor designs: one with blocking loads and no stream buffers, known as the *unenhanced design* ("un"), one with non-blocking loads ("nbl"), one with stream buffers ("sb"), and one with both stream buffers and non-blocking loads ("nbl+sb"). For the designs including stream buffers, we considered two types of stream buffers: those that used a unit stride and those that also included an allocation filter and dynamic stride calculator ("+fds"). These six processor design cases are listed in Table 1(a). For those designs including a stream buffer, we assume eight stream buffers each with four entries. The allocation filters and dynamic stride calculators use a table that stores the 16 last addresses that were found neither in the cache nor in the stream buffers. Since we used 32B cache lines throughout our study, the total data storage of the stream buffers was 1KB.

We studied the relative performance of each of the six processor designs under five memory systems configurations. A memory system configuration comprises a cache and an interface to the next lower level in the memory hierarchy. The organizations we considered are given in Table 1(b). The fetch spacing and latency numbers chosen correspond to a pipelined memory system. These systems are becoming more common with the use of new DRAM technologies such as synchronous DRAMs and pipelined SRAMs. The fetch latency of 8 and spacing of 1 is meant to be representative of microprocessors with two-level on chip caches. Having the backing store for the primary cache on-chip allows a relatively low latency and a very high bandwidth. The fetch latency of 32 and spacing of 8 is intended to be more representative of a system with a single-level of on-chip caching and an optional cache off-chip. For processors with clock frequencies of 200MHz, these latencies and spacings correspond to latencies of 40ns and 160ns and bandwidths of 6.4GB/sec and 800MB/sec. 6.4GB/sec should be achievable on-chip, while 800MB/sec could be easily obtained to an off-chip interface using synchronous DRAMs.

The ideal configuration assumes all data cache references hit in the cache and hence the six processor designs will all achieve the same performance. The other configurations are non-ideal and thus cache misses occur. It is the number of such misses and the time that the processor is

Bench-mark	Instructions (in millions)			Ideal IPC	Ideal CPI	Non-ideal data memory system					
						8K DM			64K 2-way		
	total	loads	cbranch			CPI	MCPI%	ld miss%	CPI	MCPI%	ld miss%
alvinn	3208	942	460	1.85	1.12	1.37	22	10.5	1.72	53	6.4
compress	173	29	16	1.76	0.66	0.96	45	22.3	1.15	74	9.0
dnasa	6858	1644	422	2.16	0.76	1.70	124	49.1	2.21	190	18.9
doduc	1042	238	74	1.90	1.17	1.35	15	10.3	1.20	3	0.5
ear	9506	203 5	890	1.76	1.03	1.08	5	2.8	1.03	0	$\ll 1$
eqntott	1774	220	189	1.92	0.59	0.64	5	5.5	0.73	14	3.5
espresso	2707	550	402	1.54	0.74	0.91	23	6.5	0.83	12	0.4
fpppp	4294	1131	83	2.45	0.54	1.98	267	11.9	1.74	222	0.1
hydro	5834	1355	328	1.84	0.92	1.28	36	19.1	1.90	107	13.1
mdljdp2	3228	381	313	1.61	1.18	1.34	14	16.7	1.27	8	2.6
mdljsp2	4953	656	88	2.23	0.69	0.76	10	7.2	0.73	5	0.9
ora	4551	90	33	1.68	1.08	1.10	2	1.8	1.08	0	~ 0
spice	23504	5297	1909	1.33	1.01	1.54	53	29.6	1.65	63	9.0
su2cor	5144	1100	147	2.48	0.50	1.13	56	36.3	1.19	138	10.0
swm	11172	2144	240	2.59	0.47	0.62	32	9.7	1.04	121	9.2
tomcatv	1084	307	14	2.50	0.54	1.10	56	24.9	1.36	152	9.0
wave	3673	694	221	2.14	0.67	0.82	15	8.9	0.77	15	1.4
xlisp	5869	1444	745	1.47	0.83	1.27	53	4.0	1.20	35	0.1

Table 2: Dynamic statistics for each benchmark simulated using an unenhanced processor and three memory system configurations: (1) *ideal*, all data references hit in the cache, (2) an 8K direct mapped cache with a fetch spacing of 1 cycle and a latency of 8, and (3) a 64 KB 2-way set associative cache with a fetch spacing of 8 cycles and a latency of 32. MCPI%, the memory CPI, is the difference between the ideal and non-ideal CPI values as a percent of the ideal CPI.

stalled that differentiates the six processor designs.

We have simulated the eighteen SPEC92 benchmarks which are listed in Table 2 along with some run-time characteristics. In the table, the columns under the heading "Instructions (millions)" give the dynamic instruction, load and conditional branch counts. Because the same object code and the same input-data sets were used for all simulations of a given benchmark, these numbers remain constant. Thus, we use the average number of cycles per instruction (CPI) as our primary performance measure.

While the numbers of instructions executed are significant, the instruction-word miss rate for each benchmark is usually less than 1%; the exceptions are *doduc* and *xlisp* with a 3% miss rate, and *fpppp* with a 19% miss rate. The next column in the table gives the ideal instructions per cycle, that is, the number of instructions issued per cycle when all stalls are ignored. Observe that the averages are significantly smaller than the maximum of four, a reflection of the scheduling rules and functional unit latencies.

The rest of the columns in the table give statistics for three different memory systems. The first system corresponds to the ideal system and the CPI values for this system are given in the column marked “ideal CPI”. In the ideal system, the number of cycles executed for a benchmark is a function of the number of: (1) instruction words executed, (2) stalls caused by functional-unit conflicts, (3) instruction cache misses, and (4) conditional branches. These factors remain constant for all memory configurations and processor designs.

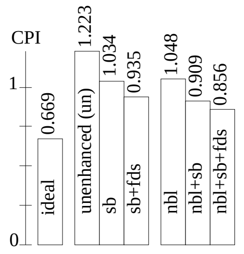
The remaining columns in the figure give statistics for an unenhanced processor using the two memory configurations noted in the caption. For a given benchmark, the portion of the CPI value due to accessing data, the memory CPI, can be found by taking the difference between its ideal CPI and the CPI values measured with a non-ideal memory system. This difference is given in the column with heading “MCPI%” as a percent of the ideal CPI. Observe that for many of the benchmarks the MCPI% value is greater than 20%. Hence, the performance of these application is significantly affected by having to access data, and therefore, it is desirable to reduce the data-access cost.

Note that this way of calculating the MCPI is valid *only* if the processor uses blocking loads. If non-blocking loads are used, a true-data dependency induced stall might be avoided because an instruction cache miss will delay the issuing of the first instruction to use the data. In other words, an instruction cache stall may allow any of the outstanding data cache misses to be resolved, resulting in fewer stall cycles being directly attributable to data references. Thus, for systems with non-blocking loads, it is not possible to determine exactly what portion of the CPI is due to accessing data. It is always true, however, that it is better to have a smaller ratio of the non-ideal to ideal CPI values.

We begin by presenting performance data for *wave* to introduce our methodology and to point out key characteristics. We then present data for all benchmarks and discuss common trends.

3.1 Common Trends

Figure 4(a) presents the CPI for *wave* measured using a memory system with an 8-Kbyte direct-mapped cache, a fetch spacing of 8 and a fetch latency of 32. In this figure, each bar corresponds to one of the six processor designs and its height reflects the measured CPI; the numbers above each bar give the actual CPI value. The figure also includes a bar representing the CPI obtained using the ideal memory system. The table in the figure gives the improvement factor for each design in relation to the unenhanced design. These factors are calculated by dividing the unenhanced CPI value by the enhanced CPI value. Note that the CPI factors for the non-blocking load and stream



(a) CPI values

design	times better
un	1.00
sb	1.18
sb+fds	1.31
nbl	1.17
nbl+sb	1.35
nbl+sb+fds	1.43

(b) CPI improvement factors

Figure 4: CPI values and ratios for *wave* using an 8 Kbyte direct mapped cache, a fetch spacing of 8 and latency of 32.

	design	miss penalty	
		large	small
CPI	un	1.233	0.819
times better	sb	1.18	1.06
	sb+fds	1.31	1.09
	nbl	1.17	1.09
	nbl+sb	1.35	1.14
	nbl+sb+fds	1.43	1.15

(a) 8K DM cache CPI improvement factors

	design	miss penalty	
		large	small
CPI	un	0.767	0.705
times better	sb	1.06	1.01
	sb+fds	1.08	1.02
	nbl	1.01	1.01
	nbl+sb	1.07	1.02
	nbl+sb+fds	1.09	1.02

(b) 64K 2-way SA cache CPI improvement factors

Table 3: CPI values and ratios for *wave* for the large miss penalty (fetch spacing 8 cycles, latency 32 cycles) and the small miss penalty (fetch spacing 1 cycle, latency 8 cycles) memory configurations.

buffer designs cannot be obtained by multiplying together the CPI factor for non-blocking loads and for stream buffers; the combined use of these techniques changes the run-time dynamics of a program.

From the table, we see that the use of unit-stride stream buffers (the "sb" design) results in a CPI improvement of 18%. When allocation filters and dynamic strides are used, this improvement increases to 31%. Non-blocking loads, on the other hand, improve the CPI by 17%, but when used with dynamic strides and allocation filters, the CPI is reduced by 43%. *From these percentages, we observe that (1) both non-blocking loads and stream buffers reduce the CPI by a minimum of 17%, (2) non-unit-stride stream buffers give better performance than either non-blocking loads or unit-stride stream buffers, and (3) non-blocking loads and non-unit-stride stream buffers together yield significantly better performance than either technique used alone.*

When the miss penalty is reduced, the performance of non-blocking loads gets better with respect to the stream-buffer-only designs. This improvement is illustrated by the data presented in Table 3(a). This table gives the unenhanced CPI and the improvement factors for the memory configuration used in Figure 4 and for a memory configuration with a smaller miss penalty. The smaller miss penalty is achieved by decreasing the fetch latency to 8 cycles and increasing the memory bandwidth (by decreasing the fetch spacing to 1 cycle). Observe that the improvement factor for non-blocking loads ("nbl" design) is equal to that for the non-unit-stride stream buffer design ("sb+fds" design) when the miss penalty is smaller. This relative improvement for non-blocking loads occurs with smaller miss penalties because it is more probable that the time required to service a cache miss will be overlapped with the execution of unrelated instructions. Observe also that the improvement factors are smaller. This fact is due to each cache miss requiring less time to be resolved and hence the miss contributes less to the number of cycles executed.

The unenhanced CPI values and improvement factors are given in Table 3(b) for a 64 Kbyte, 2-way set-associative cache and for the same two memory interface configurations. With the larger cache, we see that all the improvement factors have dropped as have the unenhanced CPI values. However, observe that same performance relationships between designs mentioned above also

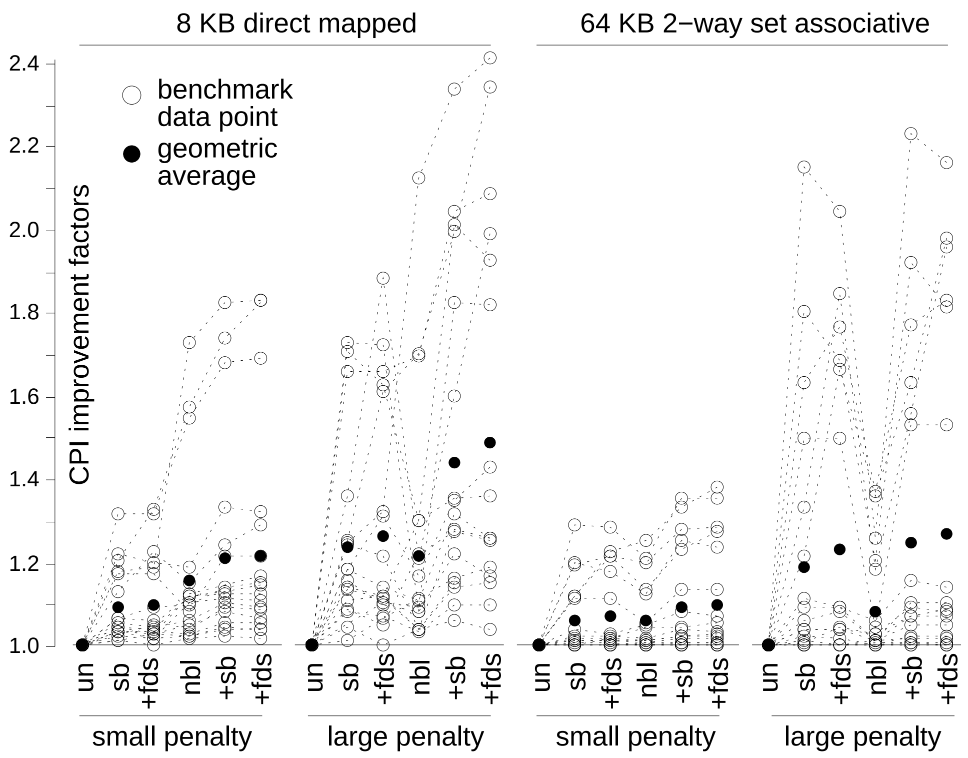


Figure 5: The CPI improvement factors for all benchmarks shown graphically; dotted lines connect points associated with the same benchmark. The “small penalty” points correspond to a fetch spacing of 1 cycle and a fetch latency of 8 while the “large penalty” points correspond to a fetch spacing of 8 and a fetch latency of 32.

apply here.

The performance relationships just discussed in the context of *wave* occur for many of the other benchmarks. The improvement factors for all 18 benchmarks for the same four memory configurations are presented graphically in Figure 5[3]. In this graph, there is an unfilled circle for each benchmark for each of the 24 machine configurations, and the filled circles give the geometric average of the improvement factors. Observe the following:

- For each memory configuration, there are a number of benchmarks that incur essentially the same CPI for all processor designs. For these benchmarks, data cache accesses contribute little towards the run-time of the benchmark. However, for other benchmarks, the choice of memory system can make a significant impact.
- Though the improvement factors for all benchmarks are quite different for a particular memory configuration, they vary in similar ways with the different processor designs. These variations are reflected in the geometric averages (the filled circles).
- Stream buffers with dynamic strides and allocation filters have somewhat larger improvement factors (i.e., better performance) than unit-stride stream buffers, although for many benchmarks the additional hardware may not be cost-effective.

[3]The data from which this graph was prepared is given in Tables 7 and 8 in Appendix B.

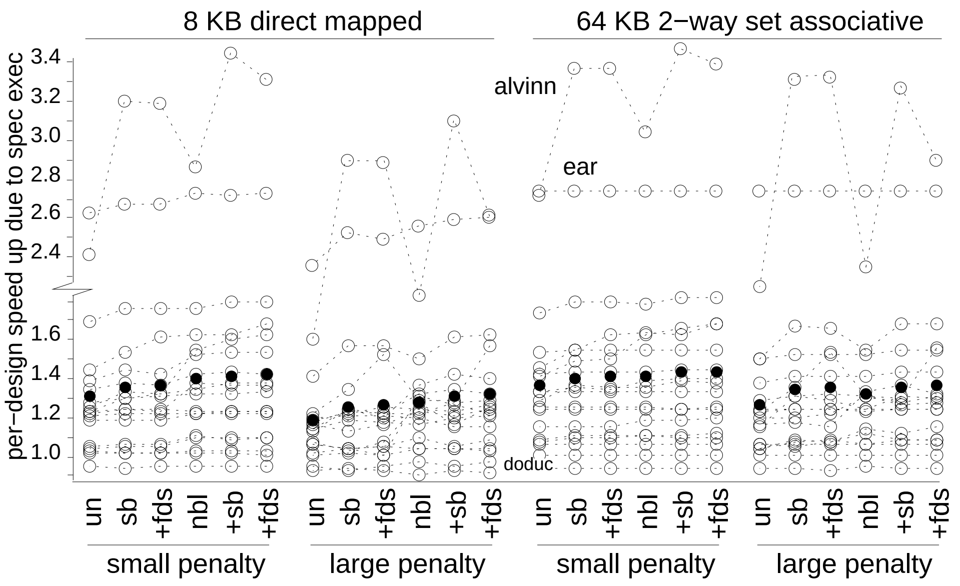


Figure 6: Run-time speedup for each processor design and memory configuration brought about by compiling the code with speculative execution. The "small penalty" points correspond to a fetch spacing of 1 cycle and a fetch latency of 8 while the "large penalty" points correspond to a fetch spacing of 8 and a fetch latency of 32.

- The combined use of stream buffers and non-blocking loads gives a significantly larger performance increase than either technique alone.
- Non-blocking loads have smaller improvement factors (i.e., are less effective) than stream buffers for the large miss penalty. With smaller miss penalties, non-blocking loads are relatively more effective.
- Non-blocking loads are relatively more effective in smaller caches than in larger ones.

Speculative Execution

The effectiveness of non-blocking loads can be increased if more unrelated instructions can be scheduled between the load and first-use instructions. Loop unrolling, and the more general notion of trace scheduling, are techniques that aid in this task by increasing the size of the basic block and hence the pool of unrelated instructions. The results we have presented so far correspond to the use of both of these techniques. Another technique is to allow the compiler to move safe instructions past branch points and thereby allow the instruction to be executed earlier. The Multiflow compiler can implement such code movements by using speculative execution.

To investigate the effect of speculative execution, we re-compiled the benchmarks with speculative execution enabled and simulated their execution on the 24 machine configurations (six processor designs times four memory configurations); speculative execution was allowed for all instructions with the exception of stores and floating-point divides. The two main observations from this investigation are that speculative execution (1) improves by 20 to 40% the run time of about half the benchmarks on all processor designs, and (2) increases the effectiveness of non-blocking loads. Data supporting the first of these observations is shown graphically in Figure 6[4]; Section 4.2 presents data for the second. This graph presents the speedup in the run-time due to speculative execution. The speedup is calculated on a per-processor-design basis by taking the run-time without speculative execution and dividing it by the run-time with speculative execution. The run-time is used here rather than CPI because speculative execution results in an increase in the number of instructions executed; this increase is discussed further in Section 4.2.

4 Understanding Performance

In the preceding section, we presented data showing that stream buffers and non-blocking loads are effective in improving the performance of a quad-issue processor. We have also seen that speculative execution increases the performance of all six processor designs. In this section, we explore further the causes for the performance gains and explain why these causes are not tied to the specific architecture we simulated.

4.1 Without Speculative Execution

We have seen that non-blocking loads, when acting alone, tend to be more effective when used with caches that have higher miss rates (e.g., smaller caches with less associativity). This is because when the miss density is higher there is a higher probability of being able to overlap more than one miss at a time.

We have also seen that stream buffers, when acting alone, tend to be more effective than non-blocking loads. For non-blocking loads, the fetching of missing data is initiated by the instruction that loads the data into a register. For stream buffers, however, the fetch can be initiated by a previous cache miss with the subsequently executed load instruction loading the register. These fetch-initiating events in effect implement the prefetching of the source operands for a dependent instruction. Non-blocking loads can be viewed as implementing *after-the-load*, or *explicit* prefetching whereas stream buffers can be viewed as implementing *before-the-load*, or *predictive* prefetching. The goal for after-the-load prefetching is to have the data needed by an instruction loaded into registers before the instruction is executed. The goal for before-the-load prefetching is to have the data nearby so that it can be quickly loaded into the registers when the load is executed.

These two types of prefetching are complementary when used together as our results have shown. When used together, the goal is the same as for after-the-load prefetching but the fetch trigger can be the load or can be a previously detected miss.

Stream buffers work well as a before-the-load prefetcher if there is a reasonable correlation between the block addresses of cache misses since stream buffers must guess what to fetch. If a correct guess is made, the reduction in the stall time is a function of the temporal separation between the fetch trigger and the execution of the load instruction for the data. In our simulations of the SPEC benchmarks with single-, dual- and quad-issue machines, we have observed two trends among the benchmarks that explain why stream buffers work. First, the cache miss addresses are

[4]The data from which this graph was prepared is given in Tables 9 and 10 in Appendix B.

bench-	s-buffer		most		2nd		3rd
mark	refs	hit%	stride	%	stride	%	stride
alvinn	6	98	32	99	128	1	124
compress	9	11	32	21	160	1	128
dnasa	24	50	32	47	1024	26	2048
doduc	0.7	5	32	91	-48	1	-8
ear	$\ll$1	71	32	99	-384	0	-192
eqntott	4	45	32	57	-192	9	384
espresso	0.4	59	32	95	64	2	-64
fpppp	0.2	8	32	50	-3200	24	-6400
hydro	13	82	32	96	832	1	1632
mdljdp2	3	57	32	94	-4000	$<$1	-8
mdljsp2	1	66	32	99	64	$<$1	-16
ora	0	51	124	59	32	40	400
spice	9	22	32	84	192	3	272
su2cor	13	67	32	89	64	1	128
swm	9	88	32	97	1160	3	1152
tomcatv	10	82	32	88	80	8	160
wave	2	69	32	81	10016	18	-210336
xlisp	0.1	67	32	95	-96	2	-88

Table 4: The most effective strides (in bytes) for each benchmark and the percent of stream buffer hits (%) corresponding to that stride. The table also gives the number of stream buffer probes as a percent of the number of loads executed (refs), and the percent of these probes that resulted in a stream buffer hit (hit%). The system configuration included non-blocking loads, a 64 Kbyte, 2-way set associative cache with 32 byte lines, and a memory interface with a fetch spacing of 8 cycles and a latency of 32 cycles.

highly predictable and thus it is likely that the correct data will be prefetched. And second, with larger caches, cache misses tend to occur at uneven intervals and in groups. This grouping results in there being time spans when no cache misses occur thereby allowing the stream buffers to fill all their entries. As a result, when a cache miss does occur, it is more likely that the needed data is already resident in a stream buffer. Hence, the stall time is much shorter.

In our simulations of the quad-issue processor, we have observed that most of the SPEC92 benchmarks achieve over a 50% hit rate in the stream buffers, an indication of the sequential nature of the miss addresses. Some of the data on which this observation is based is shown in Table 4. This table gives the stream buffer hit rates (“s-buffer hit%”) along with the percent of the executed loads that lead to stream buffer probes (“s-buffer refs”). This latter percentage is the cache miss rate; note that the miss rates given in this table are larger than those given in Table 2 due to the use of non-blocking loads here. The remaining columns in the table give the three dynamically calculated strides that accounted for most of the stream buffer hits. Notice that the unit-stride (32 bytes) is by far the most popular stride.

The preference for the unit stride suggests that the allocation filter is more important than the dynamic stride calculator. This result differs from the results reported by Palacharla and Kessler

[5] in their study of the NAS and PERFECT benchmarks. They found that some of the benchmarks obtain noteworthy improvement with the use of dynamic strides. That we do not see the same trend in our results is perhaps an artifact of the compiler we are using, or more likely, an artifact of the SPEC92 benchmarks.

The allocation filter prevents a stream buffer from being allocated when a miss occurs to a rarely used part of the address space. By preventing such allocations, prefetched data is likely to remain longer in a stream buffer and thus it is more likely to be used to satisfy a cache miss. For the machine configuration discussed above, the average percent of stream buffer prefetches that are used to resolve a cache miss increases from 22% to 47% when allocation filters are employed. Allocation filters also help by reducing the bandwidth consumed by stream buffer prefetches and thus reduce the time required to fetch data missing from the cache. For some benchmarks (e.g, *alvinn* and *tomcatv*), there is very little change in the bandwidth utilization because these benchmarks rarely access infrequently-used parts of their address space. For other benchmarks (e.g., *compress* and *dnasa*), there are many such references and the allocation filters can dramatically decrease the bandwidth utilization. The most dramatic drop occurs for *compress* where the average number of in-progress memory fetches drops from 2 to 0.5.

4.2 With Speculative Execution

With speculative execution we observed that its use (1) improves by 20 to 40% the run time of about half the benchmarks on all processor designs, and (2) increases the effectiveness of non-blocking loads. There are a number of effects that give rise to this behavior. First, the use of speculative execution gives rise to an increase in the average number of instructions issued per cycle (ideal IPC). This fact is shown in Table 5 for each benchmark by the data in the columns headed “ideal IPC”. Note that these average values apply to all 24 machine configurations because the number of instructions issued per cycle is determined at compile time. A second effect is that a given benchmark spends less time stalled due to functional-unit conflicts. This reduction occurs because with speculative execution, the compiler can schedule potentially useful instructions between the two that caused the stall rather than scheduling the stall. A third effect is that the compiler can schedule load instructions earlier. This effect is shown in the table by the increase in the weighted average number of instruction issue cycles between the load instruction and the first instruction to use the loaded value (see the columns headed “UAL distance”). Observe that while some benchmarks show greater than a 100% increase, many do not. This lack of change is in part due to our scheduling the code for cache-hit penalty rather than a larger value. The benefits of scheduling for larger values have been demonstrated by Farkas and Jouppi [16] though without the use of speculative execution. However, in quad issue machines scheduling for significantly larger latencies than the cache hit latency becomes infeasible due to a lack of sufficient registers. Due to the small change in the load-use separation, the only configurations that show a significant performance improvement when enhancing speculative execution with non-blocking loads are those using the smaller cache. This result is due to smaller caches being more sensitive to the small changes in non-blocking load dynamics.

benchmark	UAL distance		ideal IPC		benchmark	UAL distance		ideal IPC	
	without	with	without	with		without	with	without	with
alvinn	1.4	3.1	1.8	3.6	compress	1.9	1.9	1.8	2.1
dnasa	4.0	4.5	2.2	3.0	doduc	6.8	6.5	1.9	2.4
ear	2.0	9.7	1.8	3.2	eqntott	1.5	2.2	1.9	3.4
espresso	1.8	2.3	1.5	2.4	fpppp	5.1	4.9	2.4	2.6
hydro	3.3	4.2	1.8	2.5	mdljdp2	4.7	7.6	1.6	2.5
mdljsp2	5.9	6.9	2.2	2.9	ora	5.8	5.2	1.7	1.8
spice	3.0	3.0	1.3	2.1	su2cor	4.8	4.7	2.5	2.9
swm	3.2	5.4	2.6	3.7	tomcatv	5.0	5.1	2.5	2.7
wave	3.2	4.0	2.1	2.7	xlisp	1.8	2.1	1.5	1.9
average	3.2	4.2	1.9	2.6					

Table 5: The effects of with and without speculative execution. The first set of columns (UAL distance) gives the weighted average number of instruction issue cycles between a load instruction and the first instruction to use the target register. The second set of columns (ideal IPC) gives the average number of instructions issued per cycle in a machine without stall cycles.

5 Conclusions

We have investigated the relative performance impact of non-blocking loads, stream buffers, and speculative loads used individually and in conjunction with each other. We used a quad-issue microprocessor that resembles a number of state-of-the art commercial microprocessors. We have simulated 18 of the SPEC92 benchmarks and evaluated the three techniques by their ability to reduce the number of clock cycles required to execute each benchmark. An important part of this study was the use of the Multiflow Compiler Technology to compile the benchmarks. Using the compiler, we were able to generate object code that was optimized for the microprocessor architecture that we modeled, and to employ trace scheduling and speculative execution, both of which are important for wider-issue machines.

The combined use of stream buffers and non-blocking loads yields significantly better performance than is achieved with either technique acting alone. This complementary behavior is a result of stream buffers being good at reducing the cost of servicing a miss when one occurs, while non-blocking loads are good at hiding the cost of servicing a miss.

Speculative execution was found to improve the performance by 20% to 40% of processors using neither non-blocking loads nor stream buffers as well as those using one or both of these techniques. The performance gains were found to be fairly constant across all processor designs for a given cache configuration and miss penalty. This performance gain occurred because the compiler was able to schedule approximately 37% more instructions per instruction word and to reduce the number of stalls caused by functional unit conflicts. With speculative execution, the non-blocking load effectiveness increased but this increase was noticeable only with the smaller cache configuration. The non-blocking load improvement occurred because speculative execution resulted in a small increase (from 3.2 instruction issue cycles to 4.2 instruction issue cycles) in the distance between a load instruction and the first instruction to use the loaded value. This increase is nevertheless significant in view of the issue width of the processor we modeled.

The primary benefit from the use of allocation filters and dynamic strides is a reduction in the memory bandwidth consumed by prefetching. In our simulations of the SPEC92 benchmarks, we found that the SPEC92 benchmarks are dominated by unit-stride memory accesses. This observation is in contrast to the conclusion reached by Palacharla and Kessler that some of the NAS and PERFECT benchmarks favor dynamic strides.

Finally, consistent with previous studies, we have found that stream buffers are better able to tolerate larger cache miss penalties, and that the effectiveness of non-blocking loads is improved with smaller caches. This improvement appears to be caused by the higher miss rate and the higher frequency of misses.

Based on our results, the combination of speculative non-blocking loads and stream buffers can reduce the CPI incurred in a blocking system without stream buffers by an average 37% for a 64-Kbyte cache when used with a memory system that can service one request every 8 cycles and has a 32 cycle latency; with an 8-Kbyte cache, the improvement jumps to 61%. We expect that the combination of these techniques will be crucial in removing memory system bottlenecks for processors that attempt to aggressively exploit instruction-level parallelism.

6 Acknowledgments

The research described in this paper has been partially funded by the National Sciences and Engineering Research Council of Canada and by Digital Equipment Corporation.

We thank Joel Emer, Geoff Lowney, Bob Nix, and David Webb for their guidance and answers to numerous questions as we modified and used the simulation infrastructure they developed. We also thank Alan Eustace, Joel McCormack, Amitabh Srivastava, Annie Warren, and the other WRL-ites for both helping out and putting up with the simulations. Finally, we thank Digital Equipment Corporation for providing us with the Alpha AXP workstations on which we ran ≈6000 simulations, requiring ≈540 days of run-time, in 5 months.

A Stream Buffer Detail

In this appendix, we describe our implementation of the stream buffer model and the scheme we implemented for dynamically calculating stream buffer strides.

A.1 Stream Buffer Implementation

A stream buffer consists of a number of entries that are managed as a FIFO queue[4]. As illustrated in Figure 3 (page 7), each entry can store a block of data and the corresponding address (the *tag*). For each entry there is a comparator and thus all the tags can be searched at the same time. There are also a number of status flags associated with each entry whose purposes will be described below.

When a data cache reference is issued, the tags of all entries in all the stream buffers are probed at the same time as the data cache tags (although if there is a hit in the stream buffers the data is not available until the next cycle). If a data cache hit occurs, the stream buffers are left alone. If a miss should occur, one of four sequences of events can occur depending on the result of the stream buffer probe. These sequences are described below.

Data is Present

One outcome of the stream buffer probe is that the tag matched and the data is present. In this case, in the next cycle, both the cache line to which the miss address maps and the target register of the load are written; if this line is required by another load access occurring in the same next cycle, the data can be bypassed. When a hit occurs, the matching entry is shifted out of the corresponding stream buffer. Since the stream buffer is a FIFO queue, this shifting results in the invalidation of the matching entry and all entries from the entry at the head of the queue to the matching entry; invalidating an entry entails clearing the valid bit associated with the tag. This invalidation occurs *whenever* an entry's tag matches and the valid bit is set.

Because this stream buffer now has at least one invalid entry, it can begin prefetching again along the same path. The first step in initiating a prefetch is to determine the address of the next block to be fetched into the buffer. If the buffer contains no valid entries, the block address of the next block to be fetched will be one more than the tag of the matching entry. If, on the other hand, the buffer contains some valid entries, then the block address of the next block is one more than the tag contained in the bottom-most entry in the buffer.

Once a block address is determined, the next step is to check to see if the block to be prefetched is present in another stream buffer or is being prefetched. This step is necessary to ensure that the stream buffers are prefetching down non-overlapping paths. This check is performed by doing an associative search of all valid stream buffer entries. If a match is found, the stream buffer is prevented from fetching further down the present path and it is flagged to be used next when a stream buffer is needed to follow a new path (see below for a discussion on what happens when there is a stream buffer miss). As a result of ensuring non-overlapping paths, it is possible for a stream buffer to remain empty until it is reallocated on a stream buffer miss.

If the tag of the to be prefetched block is not already in another stream buffer, the tag is written into the bottom-most entry in the buffer, the valid bit for the tag is set, and a fetch request is issued to the memory. The memory assigns highest priority to fetch requests from the cache and uses round-robin arbitration to select among simultaneous requests from the stream buffers. When the memory launches a fetch request from a stream buffer, the launched bit for the entry is set. After launching a fetch request, if the stream buffer contains some invalid entries, the just described steps for initiating another prefetch are applied.

Fetch Request for Data Issued

The second outcome of the stream buffer probe is that a stream buffer has issued a fetch request for the desired block and this fetch has yet to be completed. As described above, on a hit, the matching entry is shifted out of the stream buffer. To ensure that the register is loaded when the block is returned by the memory, an MSHR is allocated. Allocating an MSHR is also necessary in order to keep track of any secondary misses that might occur before the block is written into the cache.

When a block is returned from the memory, its return may be due to either a cache-miss initiated fetch or a stream buffer initiated prefetch. In order to determine which component is awaiting its return, an associative search is done of all the valid MSHRs and all the valid stream buffer entries. If there is at least one matching MSHR, the block is written into the cache and all waiting registers are filled. Regardless of there being a matching MSHR, there may be at most one matching stream buffer entry. If there is a match, then the data is written into the entry and its present is set. Note that it is impossible for both searches to result in no matches.

Matched to an Unlaunched Entry

The third outcome of the stream buffer probe is that there is an entry with a matching tag but the memory request for this block has yet to be launched. As in the above two cases, the matching entry is shifted out of the stream buffer. And, as in the previous case, an MSHR is allocated. However, unlike the previous case, here a fetch request is issued to the memory.

Stream buffer miss

The final outcome of the probe is that there was no valid match. In this case, an MSHR is allocated and a fetch request is issued to the memory. In addition, the address of the subsequent block is calculated (as in the first outcome when there was no remaining valid entries) and a probe is done of all valid stream buffer entries. If a match is not found, a stream buffer is chosen to begin prefetching down this new path. A stream buffer is chosen from among those which were prevented from following a path (because the same path was being followed by another buffer), or if there are no such stream buffers, then we select the stream buffer with the oldest access-time stamp. The access-time stamp is set whenever a stream buffer probe results in a hit and when the stream buffer is allocated.

A.2 Allocation Filters and Non-unit Strides

A stream buffer is allocated to begin fetching a block every time a stream buffer miss occurs and the corresponding tag is not found in a stream buffer. With this allocation strategy, an isolated data reference that is not part of any address stream can cause a stream buffer to be allocated. In so doing, blocks of data will be discarded which might soon be needed in favor of prefetching down a fictions path. To prevent the first miss from allocating a stream buffer, an *allocation filter* can be used. We implement the filter proposed by Palacharla and Kessler [5]. This filter prevents a stream buffer from being allocated until two misses occur for the same stream. On the second miss, a stream buffer is allocated and it begins prefetching the block subsequent to the one corresponding to the second miss.

In the above discussion, we have assumed that the next block fetched will be the one following directly after the previous one that was fetched, that is, the block addresses of the two blocks will differ by one. Non unit-stride prefetching is more complicated as it requires first determining a stride and then determining whether to allocate a stream buffer. We implemented a scheme based on the *minimum delta* scheme proposed by Palacharla and Kessler [5]. On a stream buffer miss,

Abbrev.	Processor Details	
	blocking loads	stream buffers
un	yes	none
sb		unit stride
sb+fds		filter & dynamic stride
nbl	no	none
nbl+sb		unit stride
nbl+sb+fds		filter & dynamic stride

Table 6: Processor Designs

the allocation filter is applied to determine whether a unit-stride should be used. If there is a filter miss, then the minimum signed difference between the miss address and the last N miss addresses is determined; this minimum which may be positive or negative is the stride. In the simulations for this paper, we used $N = 16$. The allocation filter is then applied again but this time to determine whether the miss address corresponds to the third miss in a stream with a non-unit stride. If there is a filter hit, then a stream buffer might be allocated. We wait for the third miss in a non-unit stride stream to give preference to unit strides. Likewise, we check the allocation filter for a unit stride to ensure that we don't start prefetching down the same path with two different strides.

B Results for average CPI and Run-times

This appendix presents the data from which the CPI improvement factor graph (see Figure 5, page 12) and the run-time speedup graph (see Figure 6, page 13) were prepared. In presenting this data, abbreviations are used to represent each of the six processor designs. These abbreviations are given in Table 1(a) and are repeated below in Table 6.

The CPI improvement factors are given in Tables 7 and 8 for the six processor designs and the two cache miss penalties. The data in these tables corresponds to compiling the benchmarks without speculative execution.

Tables 9 and 10 present the data used to prepare the run-time speedup graph. This graph gives the run-time speedup for each processor design for and memory configuration that is brought about by compiling the code with speculative execution.

benchmark	fetch spacing of 1, latency of 8						fetch spacing of 8, latency of 32					
	unen	Times Better					unen	Times Better				
	CPI	sb	+fds	nbl	+sb	+fds	CPI	sb	+fds	nbl	+sb	+fds
alvinn	1.366	1.17	1.17	1.08	1.21	1.21	2.106	1.73	1.72	1.08	1.82	1.82
compress	0.963	1.01	1.03	1.12	1.13	1.15	1.875	1.01	1.06	1.11	1.16	1.19
dnasa	1.699	1.20	1.32	1.57	1.74	1.83	4.522	1.25	1.61	1.70	1.99	2.34
doduc	1.353	1.03	1.02	1.10	1.11	1.11	1.917	1.08	1.05	1.22	1.27	1.25
ear	1.077	1.03	1.03	1.02	1.04	1.04	1.223	1.14	1.11	1.03	1.15	1.13
eqntott	0.641	1.03	1.03	1.03	1.05	1.05	0.804	1.11	1.12	1.03	1.14	1.15
espresso	0.912	1.06	1.06	1.04	1.09	1.09	1.227	1.24	1.22	1.04	1.28	1.26
fpppp	1.983	1.05	1.04	1.12	1.13	1.13	2.737	1.14	1.12	1.30	1.35	1.36
hydro2d	1.279	1.22	1.20	1.19	1.33	1.32	2.345	1.71	1.63	1.30	2.01	1.92
mdljdp2	1.336	1.06	1.04	1.06	1.10	1.09	1.810	1.18	1.14	1.11	1.31	1.26
mdljsp2	0.762	1.04	1.03	1.05	1.08	1.07	0.989	1.15	1.10	1.08	1.22	1.17
ora	1.104	1.01	1.00	1.02	1.02	1.02	1.191	1.04	1.00	1.04	1.06	1.04
spice	1.538	1.09	1.05	1.12	1.21	1.17	3.137	1.16	1.10	1.09	1.23	1.19
su2cor	1.132	1.18	1.19	1.73	1.82	1.83	2.996	1.25	1.32	2.12	2.33	2.41
swm	0.621	1.13	1.23	1.15	1.24	1.29	1.066	1.36	1.88	1.21	1.60	1.99
tomcatv	1.105	1.32	1.31	1.55	1.68	1.69	2.799	1.66	1.66	1.70	2.04	2.09
wave	0.819	1.06	1.09	1.09	1.14	1.15	1.223	1.18	1.31	1.17	1.35	1.43
xlisp	1.275	1.02	1.02	1.02	1.04	1.04	1.512	1.08	1.07	1.03	1.10	1.09
geometric avg	1.112	1.09	1.10	1.15	1.21	1.21	1.771	1.23	1.26	1.21	1.42	1.45

Table 7: CPI improvement factors for 8K direct mapped cache.

benchmark	fetch spacing of 1, latency of 8						fetch spacing of 8, latency of 32					
	unen	Times Better					unen	Times Better				
	CPI	sb	+fds	nbl	+sb	+fds	CPI	sb	+fds	nbl	+sb	+fds
alvinn	1.268	1.11	1.11	1.05	1.13	1.13	1.717	1.50	1.50	1.04	1.53	1.53
compress	0.781	1.00	1.01	1.04	1.04	1.05	1.148	1.00	1.04	1.03	1.09	1.08
dnasa	1.121	1.12	1.23	1.25	1.33	1.38	2.211	1.22	1.66	1.36	1.63	1.98
doduc	1.173	1.00	1.00	1.00	1.01	1.01	1.199	1.01	1.00	1.01	1.02	1.01
ear	1.028	1.00	1.00	1.00	1.00	1.00	1.028	1.00	1.00	1.00	1.00	1.00
eqntott	0.622	1.02	1.02	1.01	1.03	1.03	0.726	1.09	1.09	1.01	1.10	1.10
espresso	0.814	1.00	1.00	1.00	1.01	1.01	0.831	1.02	1.02	1.00	1.02	1.02
fpppp	1.734	1.00	1.00	1.00	1.00	1.00	1.743	1.00	1.00	1.00	1.00	1.00
hydro2d	1.166	1.20	1.18	1.12	1.25	1.24	1.895	1.80	1.68	1.20	1.92	1.81
mdljdp2	1.203	1.01	1.01	1.01	1.02	1.02	1.278	1.04	1.04	1.03	1.07	1.07
mdljsp2	0.696	1.01	1.01	1.01	1.01	1.01	0.725	1.04	1.04	1.01	1.05	1.05
ora	1.075	1.00	1.00	1.00	1.00	1.00	1.075	1.00	1.00	1.00	1.00	1.00
spice	1.167	1.04	1.03	1.05	1.08	1.07	1.651	1.11	1.09	1.06	1.15	1.14
su2cor	0.681	1.19	1.21	1.20	1.28	1.28	1.192	1.63	1.77	1.37	1.77	1.83
swm	0.614	1.12	1.22	1.13	1.23	1.28	1.035	1.33	1.85	1.18	1.56	1.95
tomcatv	0.744	1.29	1.28	1.21	1.35	1.35	1.357	2.15	2.04	1.26	2.23	2.16
wave	0.705	1.01	1.02	1.01	1.02	1.02	0.767	1.06	1.08	1.01	1.07	1.09
xlisp	1.197	1.00	1.00	1.00	1.00	1.00	1.200	1.00	1.00	1.00	1.00	1.00
geometric avg	0.947	1.06	1.07	1.06	1.09	1.10	1.201	1.19	1.23	1.08	1.24	1.27

Table 8: CPI improvement factors for 64K 2-way set-associative cache.

benchmark	run-time speedup ratio due to using speculative execution											
	fetch spacing of 1, fetch latency of 8						fetch spacing of 8, fetch latency of 32					
	unen	sb	+fds	nbl	+sb	+fds	unen	sb	+fds	nbl	+sb	+fds
alvinn	2.413	3.196	3.190	2.864	3.446	3.317	1.608	2.896	2.883	1.833	3.100	2.621
compress	1.186	1.182	1.187	1.222	1.218	1.224	1.166	1.135	1.168	1.227	1.211	1.238
dnasa	1.231	1.296	1.318	1.548	1.608	1.621	1.065	1.036	1.011	1.314	1.268	1.226
doduc	0.948	0.941	0.946	0.955	0.951	0.954	0.954	0.930	0.950	0.978	0.956	0.973
ear	2.622	2.677	2.669	2.729	2.723	2.727	2.360	2.522	2.494	2.560	2.599	2.608
eqntott	1.690	1.757	1.756	1.764	1.798	1.800	1.416	1.570	1.574	1.501	1.616	1.627
espresso	1.206	1.224	1.223	1.225	1.237	1.236	1.139	1.181	1.179	1.165	1.203	1.204
fpppp	1.028	1.019	1.020	1.015	1.013	1.011	1.065	1.045	1.046	1.046	1.036	1.036
hydro2d	1.259	1.335	1.326	1.349	1.382	1.377	1.131	1.239	1.216	1.217	1.279	1.283
mdljdp2	1.341	1.376	1.368	1.423	1.431	1.428	1.176	1.230	1.225	1.320	1.347	1.327
mdljsp2	1.387	1.445	1.421	1.525	1.533	1.530	1.140	1.249	1.199	1.372	1.420	1.404
ora	1.019	1.022	1.021	1.025	1.031	1.028	0.923	0.937	0.929	0.907	0.929	0.920
spice	1.219	1.247	1.232	1.373	1.370	1.370	1.073	1.032	1.071	1.274	1.160	1.249
su2cor	1.042	1.054	1.051	1.092	1.092	1.092	1.007	1.014	1.009	1.040	1.038	1.033
swm	1.449	1.539	1.617	1.622	1.628	1.677	1.219	1.349	1.529	1.284	1.291	1.567
tomcatv	1.047	1.068	1.068	1.106	1.085	1.095	1.019	1.043	1.046	1.097	1.052	1.083
wave	1.279	1.302	1.313	1.320	1.322	1.330	1.186	1.220	1.257	1.244	1.235	1.265
xlisp	1.234	1.239	1.239	1.237	1.228	1.228	1.199	1.213	1.213	1.199	1.152	1.156
geometric avg	1.312	1.361	1.363	1.400	1.417	1.418	1.183	1.256	1.264	1.271	1.308	1.318

Table 9: Ratios of the run-time with speculative execution to the run-time without speculative execution for the 8K direct mapped cache.

	run-time speedup ratio due to using speculative execution											
benchmark	fetch spacing of 1, fetch latency of 8						fetch spacing of 8, fetch latency of 32					
benchmark	unen	sb	+fds	nbl	+sb	+fds	unen	sb	+fds	nbl	+sb	+fds
alvinn	2.713	3.367	3.367	3.044	3.472	3.388	1.874	3.315	3.326	1.970	3.270	2.898
compress	1.150	1.151	1.154	1.191	1.191	1.195	1.065	1.049	1.071	1.138	1.115	1.150
dnasa	1.408	1.489	1.497	1.621	1.664	1.681	1.159	1.178	1.153	1.335	1.369	1.322
doduc	0.941	0.943	0.942	0.944	0.945	0.944	0.937	0.937	0.933	0.945	0.942	0.940
ear	2.743	2.743	2.743	2.744	2.744	2.744	2.742	2.744	2.744	2.743	2.744	2.744
eqntott	1.733	1.794	1.791	1.786	1.816	1.816	1.497	1.669	1.663	1.549	1.683	1.684
espresso	1.241	1.243	1.242	1.242	1.243	1.243	1.232	1.239	1.239	1.234	1.240	1.239
fpppp	1.009	1.009	1.009	1.009	1.009	1.009	1.010	1.010	1.010	1.009	1.009	1.009
hydro2d	1.286	1.363	1.356	1.361	1.386	1.384	1.157	1.310	1.288	1.239	1.308	1.300
mdljdp2	1.428	1.436	1.437	1.439	1.444	1.444	1.382	1.409	1.416	1.409	1.433	1.436
mdljsp2	1.535	1.542	1.542	1.547	1.547	1.548	1.500	1.528	1.527	1.527	1.545	1.545
ora	1.060	1.060	1.060	1.060	1.060	1.060	1.060	1.060	1.060	1.060	1.060	1.060
spice	1.333	1.350	1.346	1.393	1.396	1.396	1.198	1.200	1.216	1.313	1.279	1.311
su2cor	1.086	1.106	1.108	1.107	1.113	1.114	1.044	1.076	1.088	1.068	1.083	1.084
swm	1.458	1.542	1.620	1.639	1.628	1.678	1.228	1.354	1.534	1.311	1.296	1.557
tomcatv	1.071	1.094	1.094	1.106	1.099	1.099	1.038	1.084	1.079	1.118	1.083	1.082
wave	1.328	1.334	1.336	1.335	1.330	1.330	1.292	1.312	1.324	1.305	1.270	1.275
xlisp	1.249	1.249	1.249	1.249	1.249	1.249	1.248	1.248	1.248	1.248	1.245	1.246
geometric avg	1.366	1.404	1.408	1.414	1.429	1.430	1.269	1.343	1.354	1.317	1.361	1.367

Table 10: Ratios of the run-time with speculative execution to the run-time without speculative execution for the 64K 2-way set-associative cache.

References

[1] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. *Proceedings of the 4th ASPLOS Conference*, pages 40–52, 1991.

[2] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. *Proceedings of the 5th ASPLOS Conference*, pages 62–73, 1992.

[3] Tien-Fu Chen and Jean-Loup Baer. Reducing memory latency via non-blocking and prefetching caches. *Proceedings of the 5th ASPLOS Conference*, pages 51–61, 1992.

[4] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *Proceedings of the 17th Intl. Symp. on Computer Architecture*, pages 364–373, 1990.

[5] Subbarao Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. *Proceedings of the 21st Intl. Symp. on Computer Architecture*, pages 24–33, 1994.

[6] M. Srinivas, Alexandru Nicolau, and Vickie H. Allan. An approach to combine predicated/speculative execution for programs with unpredictable branches. *In the proceedings of the International Conference on Parallel Algorithms and Compilation Techniques*, 1994.

[7] Anne Rogers and Kai Li. Software support for speculative loads. *Proceedings of the 5th ASPLOS Conference*, pages 38–50, 1992.

[8] Gurindar Sohi and Manoj Franklin. High-bandwidth data memory systems for superscalar processors. *Proceedings of the 4th ASPLOS Conference*, pages 53–62, 1991.

[9] P. Geoffrey Lowney et al. The multiflow trace scheduling compiler. *Journal of Supercomputing*, 7(1-2):51–142, 1993.

[10] S. Peter Song. Power PC 604. *In the proceedings of Hot Chips VI*, August 1994.

[11] John Edmondson and Paul Rubinfeld. An overview of the 21164 Alpha AXP microprocessor. *In the proceedings of Hot Chips VI*, August 1994.

[12] John Brennan. T5: A high-performance superscalar MIPS processor. *In the proceedings of MicroProcessor Forum*, October 1994.

[13] Anant Agrawal. UltraSPARC: A 64-bit, high-performance SPARC processor. *In the proceedings of MicroProcessor Forum*, October 1994.

[14] Scott McFarling. Combining branch predictors. *DEC WRL Technical Note TN-36*, 1993.

[15] Tse-Yu Yeh and Yale N. Patt. Alternative implementations of two-level adaptive branch prediction. *Proceedings of the 20th Intl. Symp. on Computer Architecture,* pages 124–134, May 1992.

[16] Keith I. Farkas and Norman P. Jouppi. Complexity/performance tradeoffs with non-blocking loads. *Proceedings of the 21st Intl. Symp. on Computer Architecture*, pages 211–222, 1994.

WRL Research Reports

‘‘Optimal Group Distribution in Carry-Skip Adders.’’
Silvio Turrini.
WRL Research Report 89/2, February 1989.

‘‘Precise Robotic Paste Dot Dispensing.’’
William R. Hamburgen.
WRL Research Report 89/3, February 1989.

‘‘Simple and Flexible Datagram Access Controls for Unix-based Gateways.’’
Jeffrey C. Mogul.
WRL Research Report 89/4, March 1989.

‘‘Spritely NFS: Implementation and Performance of Cache-Consistency Protocols.’’
V. Srinivasan and Jeffrey C. Mogul.
WRL Research Report 89/5, May 1989.

‘‘Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines.’’
Norman P. Jouppi and David W. Wall.
WRL Research Report 89/7, July 1989.

‘‘A Unified Vector/Scalar Floating-Point Architecture.’’
Norman P. Jouppi, Jonathan Bertoni, and David W. Wall.
WRL Research Report 89/8, July 1989.

‘‘Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU.’’
Norman P. Jouppi.
WRL Research Report 89/9, July 1989.

‘‘Integration and Packaging Plateaus of Processor Performance.’’
Norman P. Jouppi.
WRL Research Report 89/10, July 1989.

‘‘A 20-MIPS Sustained 32-bit CMOS Microprocessor with High Ratio of Sustained to Peak Performance.’’
Norman P. Jouppi and Jeffrey Y. F. Tang.
WRL Research Report 89/11, July 1989.

‘‘The Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance.’’
Norman P. Jouppi.
WRL Research Report 89/13, July 1989.

‘‘Long Address Traces from RISC Machines: Generation and Analysis.’’
Anita Borg, R.E.Kessler, Georgia Lazana, and David W. Wall.
WRL Research Report 89/14, September 1989.

‘‘Link-Time Code Modification.’’
David W. Wall.
WRL Research Report 89/17, September 1989.

‘‘Noise Issues in the ECL Circuit Family.’’
Jeffrey Y.F. Tang and J. Leon Yang.
WRL Research Report 90/1, January 1990.

‘‘Efficient Generation of Test Patterns Using Boolean Satisfiablilty.’’
Tracy Larrabee.
WRL Research Report 90/2, February 1990.

‘‘Two Papers on Test Pattern Generation.’’
Tracy Larrabee.
WRL Research Report 90/3, March 1990.

‘‘Virtual Memory vs. The File System.’’
Michael N. Nelson.
WRL Research Report 90/4, March 1990.

‘‘Efficient Use of Workstations for Passive Monitoring of Local Area Networks.’’
Jeffrey C. Mogul.
WRL Research Report 90/5, July 1990.

‘‘A One-Dimensional Thermal Model for the VAX 9000 Multi Chip Units.’’
John S. Fitch.
WRL Research Report 90/6, July 1990.

‘‘1990 DECWRL/Livermore Magic Release.’’
Robert N. Mayo, Michael H. Arnold, Walter S. Scott, Don Stark, Gordon T. Hamachi.
WRL Research Report 90/7, September 1990.

‘‘Pool Boiling Enhancement Techniques for Water at Low Pressure.’’
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Research Report 90/9, December 1990.

‘‘Writing Fast X Servers for Dumb Color Frame Buffers.’’
Joel McCormack.
WRL Research Report 91/1, February 1991.

‘‘A Simulation Based Study of TLB Performance.’’
J. Bradley Chen, Anita Borg, Norman P. Jouppi.
WRL Research Report 91/2, November 1991.

‘‘Analysis of Power Supply Networks in VLSI Circuits.’’
Don Stark.
WRL Research Report 91/3, April 1991.

‘‘TurboChannel T1 Adapter.’’
David Boggs.
WRL Research Report 91/4, April 1991.

‘‘Procedure Merging with Instruction Caches.’’
Scott McFarling.
WRL Research Report 91/5, March 1991.

‘‘Don’t Fidget with Widgets, Draw!.’’
Joel Bartlett.
WRL Research Report 91/6, May 1991.

‘‘Pool Boiling on Small Heat Dissipating Elements in Water at Subatmospheric Pressure.’’
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Research Report 91/7, June 1991.

‘‘Incremental, Generational Mostly-Copying Garbage Collection in Uncooperative Environments.’’
G. May Yip.
WRL Research Report 91/8, June 1991.

‘‘Interleaved Fin Thermal Connectors for Multichip Modules.’’
William R. Hamburgen.
WRL Research Report 91/9, August 1991.

‘‘Experience with a Software-defined Machine Architecture.’’
David W. Wall.
WRL Research Report 91/10, August 1991.

‘‘Network Locality at the Scale of Processes.’’
Jeffrey C. Mogul.
WRL Research Report 91/11, November 1991.

‘‘Cache Write Policies and Performance.’’
Norman P. Jouppi.
WRL Research Report 91/12, December 1991.

‘‘Packaging a 150 W Bipolar ECL Microprocessor.’’
William R. Hamburgen, John S. Fitch.
WRL Research Report 92/1, March 1992.

‘‘Observing TCP Dynamics in Real Networks.’’
Jeffrey C. Mogul.
WRL Research Report 92/2, April 1992.

‘‘Systems for Late Code Modification.’’
David W. Wall.
WRL Research Report 92/3, May 1992.

‘‘Piecewise Linear Models for Switch-Level Simulation.’’
Russell Kao.
WRL Research Report 92/5, September 1992.

‘‘A Practical System for Intermodule Code Optimization at Link-Time.’’
Amitabh Srivastava and David W. Wall.
WRL Research Report 92/6, December 1992.

‘‘A Smart Frame Buffer.’’
Joel McCormack & Bob McNamara.
WRL Research Report 93/1, January 1993.

‘‘Recovery in Spritely NFS.’’
Jeffrey C. Mogul.
WRL Research Report 93/2, June 1993.

‘‘Tradeoffs in Two-Level On-Chip Caching.’’
Norman P. Jouppi & Steven J.E. Wilton.
WRL Research Report 93/3, October 1993.

‘‘Unreachable Procedures in Object-oriented Programing.’’
Amitabh Srivastava.
WRL Research Report 93/4, August 1993.

‘‘An Enhanced Access and Cycle Time Model for On-Chip Caches.’’
Steven J.E. Wilton and Norman P. Jouppi.
WRL Research Report 93/5, July 1994.

‘‘Limits of Instruction-Level Parallelism.’’
David W. Wall.
WRL Research Report 93/6, November 1993.

‘‘Fluoroelastomer Pressure Pad Design for Microelectronic Applications.’’
Alberto Makino, William R. Hamburgen, John S. Fitch.
WRL Research Report 93/7, November 1993.

‘‘A 300MHz 115W 32b Bipolar ECL Microprocessor.’’
Norman P. Jouppi, Patrick Boyle, Jeremy Dion, Mary Jo Doherty, Alan Eustace, Ramsey Haddad, Robert Mayo, Suresh Menon, Louis Monier, Don Stark, Silvio Turrini, Leon Yang, John Fitch, William Hamburgen, Russell Kao, and Richard Swan.
WRL Research Report 93/8, December 1993.

‘‘Link-Time Optimization of Address Calculation on a 64-bit Architecture.’’
Amitabh Srivastava, David W. Wall.
WRL Research Report 94/1, February 1994.

‘‘ATOM: A System for Building Customized Program Analysis Tools.’’
Amitabh Srivastava, Alan Eustace.
WRL Research Report 94/2, March 1994.

‘‘Complexity/Performance Tradeoffs with Non-Blocking Loads.’’
Keith I. Farkas, Norman P. Jouppi.
WRL Research Report 94/3, March 1994.

‘‘A Better Update Policy.’’
Jeffrey C. Mogul.
WRL Research Report 94/4, April 1994.

‘‘Boolean Matching for Full-Custom ECL Gates.’’
Robert N. Mayo, Herve Touati.
WRL Research Report 94/5, April 1994.

‘‘Software Methods for System Address Tracing: Implementation and Validation.’’
J. Bradley Chen, David W. Wall, and Anita Borg.
WRL Research Report 94/6, September 1994.

‘‘Performance Implications of Multiple Pointer Sizes.’’
Jeffrey C. Mogul, Joel F. Bartlett, Robert N. Mayo, and Amitabh Srivastava.
WRL Research Report 94/7, December 1994.

‘‘How Useful Are Non-blocking Loads, Stream Buffers, and Speculative Execution in Multiple Issue Processors?.’’
Keith I. Farkas, Norman P. Jouppi, and Paul Chow.
WRL Research Report 94/8, December 1994.

WRL Technical Notes

‘‘TCP/IP PrintServer: Print Server Protocol.’’
Brian K. Reid and Christopher A. Kent.
WRL Technical Note TN-4, September 1988.

‘‘TCP/IP PrintServer: Server Architecture and Implementation.’’
Christopher A. Kent.
WRL Technical Note TN-7, November 1988.

‘‘Smart Code, Stupid Memory: A Fast X Server for a Dumb Color Frame Buffer.’’
Joel McCormack.
WRL Technical Note TN-9, September 1989.

‘‘Why Aren’t Operating Systems Getting Faster As Fast As Hardware?’’
John Ousterhout.
WRL Technical Note TN-11, October 1989.

‘‘Mostly-Copying Garbage Collection Picks Up Generations and C++.’’
Joel F. Bartlett.
WRL Technical Note TN-12, October 1989.

‘‘The Effect of Context Switches on Cache Performance.’’
Jeffrey C. Mogul and Anita Borg.
WRL Technical Note TN-16, December 1990.

‘‘MTOOL: A Method For Detecting Memory Bottlenecks.’’
Aaron Goldberg and John Hennessy.
WRL Technical Note TN-17, December 1990.

‘‘Predicting Program Behavior Using Real or Estimated Profiles.’’
David W. Wall.
WRL Technical Note TN-18, December 1990.

‘‘Cache Replacement with Dynamic Exclusion’’
Scott McFarling.
WRL Technical Note TN-22, November 1991.

‘‘Boiling Binary Mixtures at Subatmospheric Pressures’’
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Technical Note TN-23, January 1992.

‘‘A Comparison of Acoustic and Infrared Inspection Techniques for Die Attach’’
John S. Fitch.
WRL Technical Note TN-24, January 1992.

‘‘TurboChannel Versatec Adapter’’
David Boggs.
WRL Technical Note TN-26, January 1992.

‘‘A Recovery Protocol For Spritely NFS’’
Jeffrey C. Mogul.
WRL Technical Note TN-27, April 1992.

‘‘Electrical Evaluation Of The BIPS-0 Package’’
Patrick D. Boyle.
WRL Technical Note TN-29, July 1992.

‘‘Transparent Controls for Interactive Graphics’’
Joel F. Bartlett.
WRL Technical Note TN-30, July 1992.

‘‘Design Tools for BIPS-0’’
Jeremy Dion & Louis Monier.
WRL Technical Note TN-32, December 1992.

‘‘Link-Time Optimization of Address Calculation on a 64-Bit Architecture’’
Amitabh Srivastava and David W. Wall.
WRL Technical Note TN-35, June 1993.

‘‘Combining Branch Predictors’’
Scott McFarling.
WRL Technical Note TN-36, June 1993.

‘‘Boolean Matching for Full-Custom ECL Gates’’
Robert N. Mayo and Herve Touati.
WRL Technical Note TN-37, June 1993.

‘‘Circuit and Process Directions for Low-Voltage Swing Submicron BiCMOS’’
Norman P. Jouppi, Suresh Menon, and Stefanos Sidiropoulos.
WRL Technical Note TN-45, March 1994.