A load-instruction unit for pipelined processors

by R. J. Eickemeyer
S. Vassiliadis

A special-purpose load unit is proposed as part of a processor design. The unit prefetches data from the cache by predicting the address of the data fetch in advance. This prefetch allows the cache access to take place early, in an otherwise unused cache cycle, eliminating one cycle from the load instruction. The prediction also allows the cache to prefetch data if they are not already in the cache. The cache-miss handling can be overlapped with other instruction execution. It is shown, using trace-driven simulations, that the proposed mechanism, when incorporated in a design, may contribute to a significant increase in processor performance. The paper also compares different prediction methods and describes a hardware implementation for the load unit.

Introduction

In pipelined processors, stalls in the pipeline cause unwanted delays in program execution, thus causing the performance of a computing system to deteriorate. Data dependencies, cache misses, and branches are examples of events that may stall the pipeline. Since superscalar processors* [1–5] issue more than one instruction per machine cycle, a pipeline stall has more effect in relative terms on performance degradation in superscalar processors than in scalar processors. This is because for each cycle in which a pipeline is stalled, a greater number of instructions could potentially execute in a superscalar processor than in a scalar processor. This paper addresses one of the causes of pipeline stalls related to the cache access. In particular, we consider data cache fetches because they make up a significant portion of all processor instructions, especially in CISC (complex instruction-set computing) processors. Specifically, this paper describes a method of executing fetches in advance, making use of otherwise idle cache cycles.

One of the factors delaying program execution is access to storage. Caches [6] have been used to bridge the gap from the inexpensive but slow main memory to the fast processor by providing a level of storage that is intermediate in both size and cost, but with a fast access. Because of the locality in programs, keeping a relatively small but relatively fast buffer allows most references to be fast while keeping the cost affordable. When the requested data are found in the cache, the effect is a fast response,

*S. Vassiliadis, B. Blaner, and R. J. Eickemeyer, "SCISM: A Scalable Compound Instruction Set Machine," *IBM J. Res. Develop.*, submitted for publication.

typically one processor cycle. When the requested data are not in the cache, the result is a cache miss, which requires a memory access that can take many cycles. In one model of cache behavior, cache misses are divided into three types [7]:

- Compulsory: The data have not been referenced before, so must be brought into the cache.
- Capacity: The cache is not large enough to hold all of the data.
- Conflict: The cache organization prevents certain combinations of data from being in the cache at the same time.

Capacity and conflict misses can be reduced by increasing the cache size and increasing the associativity, respectively. Compulsory misses can be reduced to some extent with the inherent prefetching of larger line sizes. However, this can create more conflict misses when the line is too large. Data prefetching can reduce compulsory misses with minimal impact on conflict misses. If the software or hardware can predict which addresses will be referenced in the future, a cache miss can be handled before it is demanded; therefore, that miss does not contribute to the total execution time (i.e., the miss is overlapped). In addition, prefetching can reduce other types of misses. For example, prefetching can eliminate the miss that occurs when a line is replaced because of small capacity, and the line is later reused.

Prefetching can commonly be found in several different forms, both in software, where the compiler inserts prefetch instructions, and in hardware, where the hardware determines when to prefetch. A number of new architectures have prefetch instructions which allow the compiler to specify a data prefetch in advance. The prefetch can take the form of a special instruction that specifies the prefetch of a block of memory [8] or specifies the address of one datum, which may result in a cache-line prefetch [9, 10]. For each of these cases, the hardware must be able to process a cache miss while other instructions execute. Since the prefetch instruction does not actually use the data, e.g., store it in a register, a normal load instruction is used to access the data. Any exceptions generated by the prefetch are ignored; these resurface when the load is executed.

A design technique similar to this, but without extra prefetch instructions, is not to stall the processor on a load miss [11–13]. If there is a miss, the register for which the data are destined is marked to indicate that the register value is not valid. When the datum returns from the cache, it is stored in the register, and the register is marked valid. The processor stalls only when the loaded register is needed by a subsequent instruction. Instructions following the load are not stalled, and can complete execution. This approach has the advantage of not requiring new instructions to do the prefetch. To fully utilize this method, the distance between the load and the use of the data must be increased by having the compiler optimize the instruction scheduling. The processor design is complicated, however, by allowing instructions to complete out of order.

Another form of prefetching is special hardware in the cache to prefetch a cache line once a related cache line is referenced. For example, a cache miss results in a fetch of the requested line and also the next line, if it is not already in the cache. Other improved schemes have been proposed [6, 14]. Vector accesses and other code exhibiting regular strides are ideal for this case [15, 16]. Most or all of the prefetch proposals concentrate on numerical code, which is dominated by sequential loops or vector instructions. Often the working set of vector programs is too large to fit in a cache, so a cache is not used and prefetching is very important. In a vector processor, a vector load instruction results in a data prefetch.

An issue similar to data prefetch is that of instruction prefetch. The normal case for instruction fetching is to fetch the next sequential instruction. This is disrupted only when there is a taken branch, perhaps 10% of all the instructions. Because of this sequential pattern, instruction access patterns tend to be more regular than data access patterns. There are two common methods of branch prediction: dynamic prediction, using a branch history table (BHT) or branch target buffer [17] in hardware, and static prediction, using "hint" bits in the instruction itself. The BHT is typically a data structure, similar to a cache directory, indexed by the branch instruction address. The directory may contain one or more prediction bits and a branch target address. The prediction bits maintain the state of the branch instruction, i.e., whether it was last taken or not taken. When a branch instruction is detected by the hardware, the BHT is consulted to determine the prediction. For example, Blaner et al. [18] describe a branch unit that scans an instruction buffer looking for branches. As soon as one is found, it is predicted, and if it is predicted to be taken, the taken path is fetched and the sequential path is discarded. Since the branch is processed early and asynchronously, an instruction cache miss can often be overlapped with other instruction executions. In addition, branches that do not require computation or do not modify registers do not appear in the regular processor pipeline; therefore, they execute in zero machine cycles [18]. Similar approaches are used in other processors with a separate branch/instruction-fetch unit [3]. Dynamic branch prediction accuracy can often exceed 80% [17]. Static prediction schemes can be implemented similarly, but without the hardware required for a BHT. In this case, the predicted branch direction is specified in the instruction. The normal disruption of fetching by taken branches can be greatly reduced with branch prediction.

In this paper, a load-instruction unit applicable to general processor designs is described. The load unit is similar to the SCISM branch unit [18] in that the load address is predicted dynamically, cache hits take zero processor cycles, and cache misses can be partially or completely overlapped. Since the frequency of load instructions is usually higher than that of branch instructions, and data cache miss rates are usually higher than the miss rates of a similarly organized instruction cache, prefetching data can offer a substantial performance benefit. Store instructions are not addressed by this technique, since they are often handled using a store queue and do not stall the processor on a miss.

The paper is organized as follows. First, the general concepts of the load unit are described. Second, we report several experiments performed to evaluate different options for predicting data addresses. Consequently, a detailed performance study is reported, in order to estimate the performance benefits. Finally, a load unit design based on the results of the performance experiments is presented. In this paper the IBM ESA/390™ [19] architecture is assumed, but the techniques can be applied to other architectures as well.

Load address prediction

A load instruction typically operates as follows. A virtual address is generated, denoted by AGEN, by adding one or two registers and a constant displacement. In a cache using real addresses, the virtual address is translated to a real address which is used to access the cache. (This process can be accelerated by using techniques such as translation prediction [20], or avoided by allowing the cache to use virtual addresses [21].) The cache directory is checked, and if the addressed data are found in the cache (a hit), the data are returned to the processor. When the data are not found in the cache (a miss), the cache sends the request to memory or to another level of cache to fetch the data. A cache miss typically takes several cycles, and it delays instruction execution.

The load-instruction processor proposed in this paper is intended to improve the performance of processors by detecting and processing load instructions early. An overview of the load unit is shown in **Figure 1**. (We describe the experiments used to determine the performance of the design before giving the design details.)

Some information about the design is necessary in order to understand the performance experiments. The general concept is that the processor maintains a buffer of prefetched instructions. Load instructions are detected in this instruction buffer before they would normally begin execution in the processor pipeline(s). The load unit predicts the data address and issues a fetch from the data cache. The predicted address and the data are saved in another buffer, called the load queue, until the other

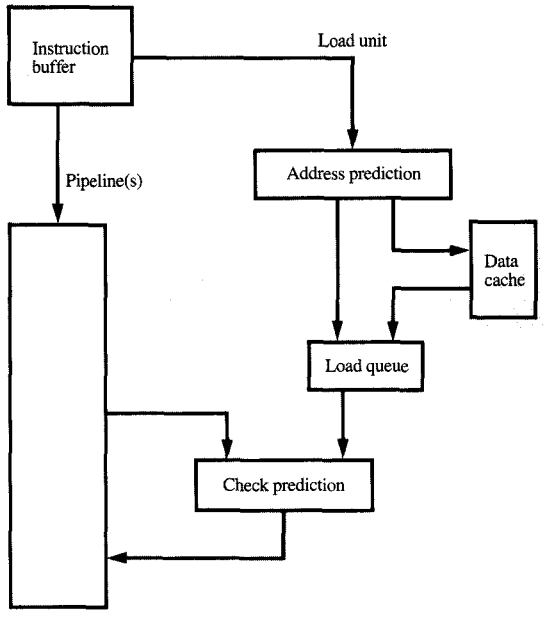

Figure 1
Basic load unit to predict data addresses and prefetch data.

instructions advance in the pipeline(s) to the point where the load instruction would normally compute its address. The address is then computed and compared to the predicted address. If the predicted address is correct, the load queue immediately supplies the data. This allows the processor to store the data in a register without waiting for the cache to provide the data. In the event of a cache miss, the data are fetched using the predicted address, a prefetch. If the prediction is made early enough, the miss is overlapped with other instructions and the processor receives the data as if there were a cache hit.

One of the impediments of the proposed load unit relates to pipeline hazards—in particular, to register dependencies. When an instruction modifies a register that is used by a later instruction, that later instruction cannot be executed until the register has been modified. When such a register is used to compute an address, the condition is termed an address-generation interlock (AGI). In a typical pipelined processor with one pipeline, an AGI can occur between two consecutive instructions, when one instruction modifies a register and the second computes an address with that register. When a load is processed early, it is likely that the register used in AGEN has yet to be modified. The modification could occur several instructions

Table 1 Instruction counts in program traces.

TSO	*Count*	*AGI*	*Source*
Pure load	216626	64152	88057
RX-format load	108270	25126	27432
SS-format load	242546	76292	2
Total load	567442	165570	115491
Stores	138627	0	0
ALU—modify regs	318435	0	49707
Other	338633	0	372
Total instructions	1363137	165570	165570

IMS	*Count*	*AGI*	*Source*
Pure load	245119	82407	97620
RX-format load	98834	28149	20055
SS-format load	223199	68394	2
Total load	567152	178950	117677
Stores	135096	0	0
ALU—modify regs	310119	0	61266
Other	336978	0	7
Total instructions	1349345	178950	178950

CICS	*Count*	*AGI*	*Source*
Pure load	231445	69057	94734
RX-format load	136930	24378	19796
SS-format load	246001	80197	0
Total load	614376	173632	114530
Stores	131325	0	0
ALU—modify regs	269929	0	59095
Other	346993	0	7
Total instructions	1362623	173632	173632

RAMPC	*Count*	*AGI*	*Source*
Pure load	153058	64858	91244
RX-format load	79701	24070	14286
SS-format load	342387	107109	568
Total load	575146	196037	106098
Stores	98097	0	0
ALU—modify regs	258604	0	89939
Other	429753	0	0
Total instructions	1361600	196037	196037

FPC1	*Count*	*AGI*	*Source*
Pure load	226671	61412	63857
RX-format load	99637	20934	28190
SS-format load	256855	170105	0
Total load	583163	252451	92047
Stores	203224	0	0
ALU—modify regs	394169	0	151124
Other	320349	0	9280
Total instructions	1500905	252451	252451

before the load instruction, not just in the previous instruction. Because load data addresses can vary and the number of AGIs can be large, the address-prediction algorithm is the key step in prefetching load data.

Given that the success of early load-instruction processing can be directly related to the frequency of AGIs, it is imperative to answer the following two questions:

1. How often does the address of a preprocessed load instruction exhibit an AGI?
2. If the frequency of AGIs for preprocessed loads is high, can the load address be predicted in a general computing environment using a mechanism that is relatively simple to implement?

In the sections that follow, these questions are answered, beginning with the evaluation methodology and an investigation of program characteristics with regard to AGIs.

• *Evaluation methodology and program characteristics*

In order to answer the previously stated questions, the evaluation uses the ESA/390 architecture as the example architecture. Data were collected from several IBM ESA/390 instruction traces. A SCISM* processor performance model was modified to detect load instructions early and collect statistics on prediction algorithms and the cache miss rates of prefetches. The performance model models the complete processor, including instruction latencies, cache misses, branch prediction, and cache trailing-edge effects. For this paper, the load instructions in the ESA/390 architecture are divided into three types. The "pure" loads (L—load and LH—load halfword) load a register without using the previous contents of the loaded register; RX-format loads (IC—insert character, A—add, C—compare, SH—subtract halfword, etc.) produce a result in a register by combining the contents of the register with data from storage; and SS-format loads and other instructions (MVC—move character, CLI—compare logical immediate, AP—add decimal, TM—test under mask, etc.) load from storage but do not modify registers. Most instructions in the SS-format load category typically load from one operand (operand 2) and store to another operand (operand 1), but some of the SS-format loads instead load from both operand 1 and operand 2. In this paper, only the first word of operand 2, the usual source, is considered for prefetching. Subsequent words, when operand 2 specifies a long block of bytes, are not prefetched explicitly, but since some of the other bytes are in the same cache line, they can also be prefetched. All

*S. Vassiliadis, B. Blaner, and R. J. Eickemeyer, "SCISM: A Scalable Compound Instruction Set Machine," *IBM J. Res. Develop.*, submitted for publication.

of the load instructions, as categorized above, perform a common step, namely: Compute an address and fetch from memory. Some instructions store the result in a register, some use it for further computation, some store it in memory, and some perform other memory operations. Other instructions are divided into three more types: stores (ST—store, STC—store character, MVI—move immediate, etc.), ALU operations, including branches that modify registers (AR—add register, SLL—shift left logical, BCTR—branch on count, etc.), and other instructions (branches not modifying registers, RR-format floating-point instructions, very complex instructions, and privileged instructions).

Data were collected from several program traces run on IBM ESA/390 architecture:

- TSO: representative workload, MVS SP 2.1.7.
- IMS: hierarchical database running on MVS SP 2.2.
- CICS™: transaction processing running on MVS SP 2.1.7.
- RAMP-C: transaction processing benchmark.
- FPC1: engineering workload.

Tables 1 and **2** show statistics gathered from the traces; both raw counts and frequencies are shown. The tables show the instruction frequencies for the three categories of loads and the other three instruction categories. Loads make up just over 40% of the executed instructions. The third column of Table 1 and the second part of Table 2 show the number of AGIs for each type of instruction. For each load-instruction category, the number of AGIs is indicated. (No AGIs are shown for other instructions, since they are not relevant to this paper.) The number of AGIs is measured by assuming that the load data address is computed early, by detecting the load in the instruction buffer. This is similar to early branch-instruction detection in the SCISM branch unit [18]. How early the address is computed depends on instruction buffer size, branch-prediction rate, instruction cache miss rate, and pipeline stalls. Table 2 shows that an average of about 33% of loads have AGIs. Slightly less than half of these are caused by the instructions immediately preceding the load. The high number of AGIs indicates that early load processing cannot be ensured by using registers alone. The last column of Table 1 and the bottom section of Table 2 also show which instructions are the source of the AGI, i.e., which instructions modify a register used subsequently to compute the load address. The majority of load AGIs are caused by other load instructions, with a large number of ALU instructions also causing AGIs. The AGI sources from SS-format loads are due to the TRT instruction which implicitly modifies a register.

Since the frequency of AGIs is high, handling only non-AGI cases prevents many load addresses from being computed early. This suggests that predicting the data addresses may be necessary. The following are the results of experiments to compute the data addresses for load instructions. The methods involve various combinations of blind AGEN, detecting AGIs, and address prediction.

Table 2 Instruction frequencies in program traces.

	TSO	*IMS*	*CICS*	*RAMPC*	*FPC1*
	Instruction type distribution (%)				
Pure load	15.9	18.2	17.0	11.2	15.1
RX-format load	7.9	7.3	10.0	5.9	6.6
SS-format load	17.8	16.5	18.1	25.1	17.1
Total load	41.6	42.0	45.1	42.2	38.9
Stores	10.2	10.0	9.6	7.2	13.5
ALU—modify regs	23.4	23.0	19.8	19.0	26.3
Other	24.8	25.0	25.5	31.6	21.3
	Load AGIs per instruction (%)				
Pure load	29.6	33.6	29.8	42.4	27.1
RX-format load	23.2	28.5	17.8	30.2	21.0
SS-format load	31.5	30.6	32.6	31.3	66.2
Total load	29.2	31.6	28.3	34.1	43.3
	Load AGI source distribution (%)				
Pure load	53.2	54.6	54.6	46.5	25.3
RX-format load	16.6	11.2	11.4	7.3	11.2
SS-format load	0.0	0.0	0.0	0.3	0.0
Total load	69.8	65.8	66.0	54.1	36.5
Stores	0.0	0.0	0.0	0.0	0.0
ALU—modify regs	30.0	34.2	34.0	45.9	59.9
Other	0.2	0.0	0.0	0.0	3.7

For the purpose of summarizing the concepts, results are presented for a fixed hardware configuration unless otherwise noted. The modeled system is assumed to have separate instruction and data first-level caches and TLBs. The second-level cache is shared. Except when indicated, the data TLB has 256 entries, is two-way associative, and supports 4096-byte pages. The first-level data cache is 64 kilobytes (KB), is four-way associative, and has 64-byte lines. The second-level cache is 512 KB, is eight-way associative, and has 128-byte lines. There is a BHT containing 1024 entries, an instruction cache of 32 KB, and a 24-byte instruction buffer, which holds approximately six instructions.

- *Load address prediction by detecting AGIs*

In describing the various experiments, the format of the results of each experiment is shown in an example. The

Table 3 Prediction using AGEN.

TSO	*TLB miss* (%)	*L1 hit* (%)	*L2 hit* (%)	*L2 miss* (%)	*Total* (%)
Correct	0.38	67.29	1.92	1.23	70.82
Incorrect	6.29	18.38	2.72	1.78	29.17
Total predicted	6.67	85.67	4.64	3.01	100.00
No prediction 0.00%					

simplest method of predicting the correct data address is to compute the address with whatever values happen to be in the registers when the load instruction is detected. This prediction is correct if the registers are not modified before the actual AGEN occurs, i.e., there is no AGI. **Table 3** shows the results using TSO as an example. The various possible results are divided into several categories:

- TLB miss ("TLB miss").
- TLB hit and first-level cache hit ("L1 hit").
- TLB hit, first-level cache miss, and second-level cache hit ("L2 hit").
- TLB hit, first-level cache miss, and second-level cache miss ("L2 miss").
- No prediction.

The results indicate how many predictions were correct or incorrect and what level in the memory hierarchy was used before the predicted address was satisfied. The predicted virtual address is first converted to a real address by consulting the TLB. In Table 3, 6.67% of the references miss the TLB. Because few TLB miss predictions are correct (0.38%), and because TLB miss handling is expensive in machine cycles, no further processing is performed on these references until the correct address is known. Of the remaining references, 85.67% hit in the L1 cache; 67.29% of the total were correct, and the correct data are prefetched, while 18.38% were incorrect. This leaves 7.65% of the references that miss L1 and reference L2; 4.64% are L2 hits, although the majority are incorrect predictions. Of the total references, 3.01% are L2 misses; again the majority are incorrect. It should be noted that the sum of all the entries in the "incorrect" line is equal to the AGI count in Table 2. Of all the L1 misses, however, most are likely to be wrong predictions. Prefetching on L1 misses may not be advisable in this case. For some prediction methods, certain cases are assumed to be probably wrong, and result in no prediction. If the prediction is expected to be wrong, it is better to wait than to create extra memory traffic.

An improvement in the prediction can be made if AGIs are detected. If the AGI is known, the address is almost certainly wrong, and no prediction should be made. However, load instructions are processed in a special unit. It is desired to process the loads one per cycle when possible. Since the load unit is detecting load instructions, other nonload instructions can be skipped. Consequently, if there is an AGI between an ALU instruction and a load, the ALU instruction is not normally fully decoded, and the AGI is not detected. Since most of the AGIs are caused by loads, which are handled by the load unit, these AGIs can be detected. If a previous load has been predicted but not yet resolved, i.e., is still waiting to see whether the predicted address is correct, the register modified by this load is available to compare to newly predicted loads and detect an AGI. This technique detects most of the AGIs. The following instruction sequence shows an AGI caused by an ALU instruction which modifies register R1:

```
L    R6, 50(R2, R3)   ; Load R6 from mem(R2 + R3 + 50)
AR   R1, R6           ; Add R1 = R1 + R6
L    R9, 20(R1, R4)   ; Load R9 from mem(R1 + R4 + 20)
```

The AGI in the second load instruction is not detected, since the load unit does not decode the add instruction. However, if the first load instruction loaded register R1 instead of R6, the AGI in the second load would be detected. Consequently, there are a few cases where an AGI caused by a nonload instruction is detected. The address-generation step for this algorithm consists of reading the registers and checking for unresolved loads that modify the needed registers. If there is no conflict, assume that the current registers are valid, generate the address, and issue the data fetch. **Table 4** shows the results of this algorithm. About 20% of the time no prediction is made because of a detected AGI, which is only slightly higher than the number of load-caused AGIs in Table 2. The number of correct predictions is the same as when addresses are computed without checking for AGIs. It should be noted that the number of wrong predictions is greatly reduced.

- *Load address prediction using deltas*

While detecting AGIs can eliminate many wrong predictions, it does little to increase the number of correct predictions. Many programs, especially scientific ones, traverse arrays or other regular data structures. Since this is usually accomplished by looping through the same code, there is likely to be some repeated pattern which can be detected and used to predict future accesses. Furthermore, it is likely that in general-purpose computing, memory locations are reserved for global or temporary values such that the address does not change. The patterns form the basis for a family of history-based prediction algorithms, called delta algorithms, which attempt to detect addressing patterns used by individual load instructions.

Each delta algorithm uses a load delta table (LDT), which is a mechanism similar to a BHT. In the LDT, the address of the load instruction is used to access the table.

Figure 2 shows the formats for the algorithms discussed in this paper. In addition to what is shown in the figure, each entry contains an address tag for matching with a table-lookup address and typical management information, such as valid bit and replacement algorithm information. In this section, the LDT is assumed to contain 4K entries with a four-way set-associative organization.

The simplest load instruction to predict is one which always (or nearly always) loads from the same address (delta is zero). In this case, a BHT-like structure will correctly predict the address. Each entry in the LDT consists of a target address, which is the data address most recently used for the instruction represented by the table entry. The process of accessing the LDT is as follows. The instruction address of the load (virtual address) is used to access a table. The lower address bits select the entries in the table, and the upper bits are matched against saved upper bits for table entries, as in a BHT or a cache directory. When there is a match, the target address is read from the table entry and is used as the predicted address. If there is no match, no prediction is available. To update the table, a new entry is created on a miss, or the target address is modified if it changed, or nothing is done if the predicted address was correct. Experiments with this algorithm indicate that in TSO approximately 50% of addresses can be correctly predicted [22].

An improved algorithm assumes a constant delta. Each entry in the LDT contains a target address and a delta value. The target address is the last address used by this load instruction. Delta is the difference between the last address and the penultimate address. When an entry is first created, the delta is set to zero, since zero is the most common delta used, and a prediction to the same address will almost always be a TLB and a cache hit. To predict an address, the LDT is searched; if there is a hit, the target and delta are added to form the predicted address. (If there is not enough time to perform this add in an implementation, the LDT can hold the target, delta, and next predicted address. The next predicted address is computed when the table is updated, when an extra cycle is presumably not as critical.) If there is a miss, no prediction is made. The LDT update process consists of creating a new entry on a miss, with delta of zero. If there was a hit, the target address is updated, and the new delta is computed (if it changed) by subtracting the target address from the actual address. This algorithm predicts approximately 59% of addresses correctly in TSO [22].

Several other algorithms were studied on the basis of observed addressing patterns [22, 23]. In this paper, an algorithm is described that performs well at reasonable cost. The algorithm uses an LDT entry consisting of a target address, two delta fields (delta1 and delta2), and one bit indicating that the entry was recently initialized. The algorithm is depicted using C-like code in **Figure 3**. This algorithm does not change the delta every time there is a wrong prediction, but only when the prediction is wrong and the last two (wrong) deltas are the same. When an

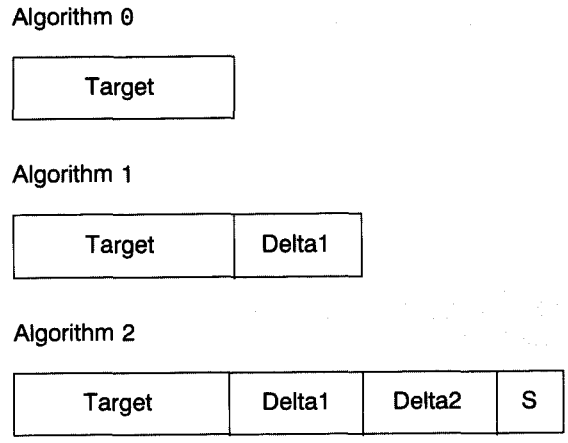

Figure 2
Delta algorithm table fields.

Table 4 Prediction using AGEN with AGI detection.

	TLB miss (%)	*L1 hit* (%)	*L2 hit* (%)	*L2 miss* (%)	*Total* (%)
TSO					
Correct	0.38	67.29	1.92	1.23	70.82
Incorrect	1.15	5.26	0.67	0.32	7.40
Total predicted	1.53	72.55	2.59	1.55	78.22
No prediction 21.78%					
IMS					
Correct	0.45	65.67	1.72	0.61	68.45
Incorrect	1.30	5.91	1.14	0.72	9.07
Total predicted	1.75	71.58	2.86	1.33	77.52
No prediction 22.50%					
CICS					
Correct	0.27	68.99	1.65	0.83	71.74
Incorrect	0.83	6.53	0.82	0.37	8.55
Total predicted	1.10	75.52	2.47	1.20	80.29
No prediction 19.71%					
RAMPC					
Correct	0.14	63.76	0.98	1.03	65.91
Incorrect	0.22	13.42	0.54	0.36	14.54
Total predicted	0.36	77.18	1.52	1.39	80.45
No prediction 19.55%					
FPC1					
Correct	0.03	52.61	0.92	3.15	56.71
Incorrect	0.75	20.21	0.57	3.37	24.90
Total predicted	0.78	72.82	1.49	6.52	81.61
No prediction 18.38%					

```
/* Representation of the correct LDT entry */
struct {
        int TargetAddress : 32;
        int Delta1 : 8;
        int Delta2 : 8;
        int Initial :  1;
        } LDTentry;

/* prediction function; add target and delta1 */
LDTprediction (int LoadInstrAddr)
{
        if (LDThit(LoadInstrAddr))
                return(LDTentry.TargetAddress + LDTentry.Delta1);
        else
                return(MISS);
}

/* update function: update target and delta2,
                    update delta1 if delta2 matches actual delta */
LDTupdate (int LoadInstrAddr, int ActualAddress, int HitMiss)
{
        if (HitMiss == MISS)
        {
                allocateLDTentry(LoadInstrAddr);
                LDTentry.TargetAddress = ActualAddress;
                LDTentry.Delta1 = 0;
                LDTentry.Delta2 = 0;
                LDTentry.Initial = 1;
        }
        else  /* HIT */
        {
                int ActualDelta = ActualAddress - LDTentry.TargetAddress;
                if (LDTentry.Initial == 1)
                {
                        LDTentry.Delta1 = ActualDelta;
                        LDTentry.Initial = 0;
                }
                else  /* is not the initial entry */
                {
                        if (ActualDelta == LDTentry.Delta2)
                                LDTentry.Delta1 = ActualDelta;
                }
                LDTentry.TargetAddress = ActualAddress;
                LDTentry.Delta2 = ActualDelta;
        }
}
```

Figure 3

Two-delta address-prediction algorithm.

entry is created, the target address is entered, both deltas are set to zero, and the initial bit is one. A prediction is the sum of the target address and delta1. On a correct prediction, the target address is updated, but the other fields are not changed. On an incorrect prediction, if the entry was initialized on its most recent access, delta1 is set to the actual delta that occurred. The target address is updated and delta2 is set to the actual delta. This algorithm is successful when the access pattern is usually a constant delta but there are occasional breaks in the sequence. For example, when the processor is stepping through an array and comes to the end, there is a jump back to the beginning or to another array. The sequence of addresses in these cases might be 100, 104, 108, 112, 100, 104, 108, 112, 100, · · · and 100, 104, 108, 112, 200, 204, 208, 212, 300, · · · . The two-delta method mispredicts the jump in the sequence but does not change the delta used for prediction, and it will be right on the next address prediction. An algorithm with a single delta mispredicts the jump and also mispredicts the next address after the jump. Another accessing pattern is to alternate between two addresses—for example, 100, 104, 100, 104, · · · . With a single delta, all predictions are incorrect, but with two deltas, half the predictions are correct.

The first part of **Table 5** shows the results using this method. The LDT organization used was 4K entries and four-way set associative. The delta fields are each eight bits. Overall, the prediction is correct just over 63% of the time. The majority of incorrect predictions occur when no prediction is made because of an LDT miss.

• *Load address prediction using LDT and AGI detection*
The prediction using LDT can be improved substantially when it is combined with a normal AGEN and LDT is used to track AGI history. The second part of Table 5 shows results when LDT is accessed in parallel with a normal AGEN. If there is an LDT hit, that prediction is used; if it is a miss, the AGEN result is used. Now 79% of addresses can be correctly predicted.

Finally, the third part of the table shows the case where LDT tracks AGI history. This optimization was introduced for branches in a BHT organization [24]. For the load unit, the prediction uses both AGEN and LDT. If the prediction was made using AGEN (an LDT miss) and it was correct, no entry is created in LDT. If the AGEN prediction was incorrect because of an AGI, an LDT entry is created. Thus, the only LDT entries correspond to instructions that once had an AGI. This reduces the size of the LDT needed. Since the same instruction sequences may occur more than once, if there is an AGI one time, there will probably be an AGI the next time the same sequence is executed. An LDT hit indicates that an AGI is very likely for this instruction (whether caused by a load or another instruction); therefore, the next best method to AGEN, the LDT, should be used. The third part of Table 5 shows results using this method. Prediction accuracy is now close to 87%.

The final improvement on this algorithm is to detect AGIs caused by load instructions and to make no prediction when the AGI is detected and there is an LDT miss. **Table 6** shows the results for each of the benchmark

Table 5 Prediction using LDT and AGEN.

TSO	*TLB miss* (%)	*L1 hit* (%)	*L2 hit* (%)	*L2 miss* (%)	*Total* (%)
LDT only					
Correct	0.13	61.38	0.85	0.79	63.15
Incorrect	0.82	13.41	0.50	0.42	15.15
Total predicted	0.95	74.79	1.35	1.21	78.30
No prediction 21.71%					
LDT and AGEN on miss					
Correct	0.41	75.33	1.92	1.32	78.98
Incorrect	2.38	16.43	1.20	1.02	21.03
Total predicted	2.79	91.76	3.12	2.34	100.00
No prediction 0.00%					
LDT and AGEN, table entry only if AGI					
Correct	0.51	82.40	2.34	1.51	86.76
Incorrect	1.72	9.83	0.88	0.81	13.24
Total predicted	2.23	92.23	3.22	2.32	100.00
No prediction 0.00%					

Table 6 Prediction using LDT, AGEN, and AGI detection (LDT and AGEN, LDT entry only if AGI, detect load–load AGI).

TSO	*TLB miss* (%)	*L1 hit* (%)	*L2 hit* (%)	*L2 miss* (%)	*Total* (%)
Correct	0.51	82.40	2.34	1.51	86.76
Incorrect	1.01	8.13	0.55	0.50	10.19
Total predicted	1.52	90.53	2.89	2.01	96.95
No prediction 3.05%					
IMS					
Correct	0.63	82.10	2.33	0.83	85.89
Incorrect	1.36	9.73	0.99	0.44	12.52
Total predicted	1.99	91.83	3.32	1.27	98.41
No predicted 1.59%					
CICS					
Correct	0.45	83.99	2.22	1.00	87.66
Incorrect	0.80	9.48	0.86	0.19	11.33
Total predicted	1.25	93.47	3.08	1.19	98.99
No prediction 1.01%					
RAMPC					
Correct	0.24	82.72	1.87	1.14	85.97
Incorrect	0.61	11.63	1.23	0.29	13.76
Total predicted	0.85	94.35	3.10	1.43	99.73
No prediction 0.27%					
FPC1					
Correct	0.04	84.04	3.14	5.66	92.88
Incorrect	0.03	5.55	0.36	0.87	6.81
Total predicted	0.07	89.59	3.50	6.53	99.69
No prediction 0.31%					

Table 7 Varying the LDT configuration (LDT and AGEN, LDT entry only if AGI, detect load–load AGI).

TSO

Size = 1024, associativity = 2, delta size = 32 bits

	TLB miss (%)	*L1 hit* (%)	*L2 hit* (%)	*L2 miss* (%)	*Total* (%)
Correct	0.44	81.19	2.30	1.75	85.68
Incorrect	0.62	7.24	0.36	0.38	8.60
Total predicted	1.06	88.43	2.66	2.13	94.28
No prediction 5.72%					

Size = 1024, associativity = 2, delta size = 8 bits

	TLB miss (%)	*L1 hit* (%)	*L2 hit* (%)	*L2 miss* (%)	*Total* (%)
Correct	0.42	81.08	2.18	1.45	85.13
Incorrect	0.61	7.87	0.36	0.29	9.13
Total predicted	1.03	88.95	2.54	1.74	94.26
No prediction 5.72%					

Size = 4096, associativity = 4, delta size = 32 bits

	TLB miss (%)	*L1 hit* (%)	*L2 hit* (%)	*L2 miss* (%)	*Total* (%)
Correct	0.52	82.50	2.46	1.80	87.28
Incorrect	1.04	7.42	0.56	0.64	9.66
Total predicted	1.56	89.92	3.02	2.44	96.94
No prediction 3.05%					

Size = 4096, associativity = 4, delta size = 8 bits

	TLB miss (%)	*L1 hit* (%)	*L2 hit* (%)	*L2 miss* (%)	*Total* (%)
Correct	0.51	82.40	2.34	1.51	86.76
Incorrect	1.01	8.13	0.55	0.50	10.19
Total predicted					
No prediction 3.05%	1.52	90.53	2.89	2.01	96.95

programs. The prediction rates for TSO are the same as before, but 3% of the formerly wrong predictions now result in no prediction. Prediction accuracy for the benchmarks ranges from about 86% to nearly 93%. In most cases, on a TLB miss the address is usually wrong, and the miss should not be processed. In going to L2 after an L1 miss, the address is correct more than two thirds of the time for some programs and nearly 90% of the time for FPC1. These results indicate that prefetching L1 misses is likely to result in a performance improvement. Fetching L2 misses also most often brings in the correct data. Recall that about 33% of loads have an AGI, and about 19% of loads have an AGI caused by another load. With the combination AGI detection, AGEN, and LDT, only 1% of loads result in no prediction. The LDT is effectively used as an AGI history indicator, since load instructions with AGIs are put in the LDT and the LDT subsequently provides the prediction. Because of these facts, it is observed that the majority of predictions are made using a computed address rather than using the LDT.

- *LDT configurations*

Thus far, performance results have not considered LDT size. **Table 7** shows results for TSO using the combined algorithm of LDT, AGEN, and AGI detection. The LDT configuration is varied. The first two parts show a smaller LDT of 1K entries. The larger LDT results in about 1.5% more correct predictions, but also makes more wrong predictions. The smaller LDT makes no prediction more often, since there are more LDT misses. The algorithm uses the LDT prediction if there is a hit, and uses the load–load AGI detection results only on a miss. Because most of the predictions are made using AGEN rather than LDT, the prediction success rate is not very sensitive to LDT size.

The table also shows how the size of the delta fields affects predictions. To reduce the cost of the LDT, the full 32-bit delta need not be used. Previous results use eight-bit delta fields. If the actual delta does not fit in eight bits, a delta of zero is used. A larger delta results in a half percent more correct predictions. A smaller delta results in more cache and TLB hits, since nearby addresses are more likely to be in the cache or TLB. More than eight delta bits results in little performance gain. The performance drops significantly when the number of bits goes below eight [22].

- *Performance improvement*

The results above suggest some implementation options for using the prediction. The majority of predicted addresses are found in both the TLB and the first-level data cache. As described below, this allows the load, which is processed early, to get its data from the cache early. On a TLB or cache miss, the processor cancels the prefetch (there is a "prefetch" signal to the TLB and cache and a "miss" signal back to the processor). Assuming that there is at least one available cache cycle between the time the cache would be accessed with the predicted address and the time the cache would be accessed when the correct address is known with certainty, the cache can be accessed early and one pipeline stage can be eliminated from normal sequential processing. Since about 40% of all instructions in the benchmarks are loads, and one cycle can be eliminated with each correct prediction on a cache hit, about 32% of instructions may execute in one fewer cycle. Therefore, the number of cycles per instruction (CPI) is expected to be reduced by 0.32.

Another policy is to prefetch data on a miss. The prediction on a TLB miss is more often wrong than right. Also, a TLB miss usually requires a large number of cycles to handle. For these reasons, if the predicted address results in a TLB miss, no further actions should be taken. When the correct address is determined, the access is attempted again. If there is still a TLB miss, it is handled then. A first-level cache miss is typically only a

few cycles in penalty. The prediction is usually correct for an L1 cache miss and an L2 cache hit. Therefore, a design option is to process the L1 cache miss early. A second-level cache miss often involves a longer penalty. Since the prediction is usually correct, L2 misses can also be processed early. To estimate the benefits of prefetching, the number of cycles between the prefetch and the fetch when there has been no prefetch was measured. This provides an indication of the number of cycles available for prefetching. On average, the prefetch happens about 12 cycles earlier than the fetch when there is no prefetch. This is enough overlap for an L2 hit, but probably not enough for an L2 miss to be completely overlapped.

The overall performance improvement, as measured in cycles per instruction (CPI), was calculated for each benchmark. The benefit calculation assumes a single-cycle access for an L1 hit, a five-cycle L1 latency for an L2 hit, and a 25-cycle latency on an L2 miss. For each load instruction, the time available for prefetching is determined by observing the number of cycles between the time when the prefetch starts and the time when the load actually executes. The average prefetch time in each case was used to determine how many cycles were saved on a correct prediction and how many cycles were lost on an incorrect prediction.

Table 8 shows the estimated CPI improvement for each of the three prefetch policies for prediction using delta table, AGEN, and AGI detection. Combinations of some or all of the three policies can be implemented. The total CPI savings range from 0.42 to 0.53. This estimate does not take into account several second-order effects that reduce the possible improvement. Among these are bus conflicts, cache utilization, stores into prefetched data, and prefetches from pending stores. It is not expected that these issues will reduce the prefetching benefit substantially. Implementation aspects are discussed below.

The benefit of actual prefetching is fairly small, because most predictions result in a cache hit in the large cache modeled. If the cache is reduced in size, prefetching is more significant. **Table 9** shows the different prediction rates for various L1 cache sizes for TSO as the L2 cache size is constant. As the cache size decreases, there are fewer hits; consequently, fewer instructions can prefetch directly from L1. The proportion of all references that reach L2 increases. Since the difference between correct and incorrect predictions that hit L2 increases substantially, it is more useful to prefetch from L2 when the L1 cache is small. **Figure 4** shows CPI improvement on TSO for different prefetch policies. The curves show the CPI relative to a 64KB cache without prefetching. For example, the upper curve shows that a 4KB cache with no prefetching has a CPI about 0.33 greater than a 64KB cache with no prefetching, and the lowest curve shows that the CPI for a 4KB cache with prefetching on L2 misses is 0.24 less than for a 64KB cache without prefetching, and that the total improvement due to prefetching for a 4KB cache is 0.57 CPI. Prefetching on an L1 hit is shown by the second curve. The prefetching improves CPI by 0.31–0.34 from a 4KB cache to a 64KB cache. The third curve adds prefetching on an L1 miss and an L2 hit, and shows further improvement from 0.22–0.05 CPI. The fourth curve includes prefetching on an L2 miss and shows improvement from 0.04–0.03 CPI. As expected, with smaller caches there are fewer hits, so there is less saving possible with the L1-hit prefetch policy. For the same reason, prefetching on an L1 miss results in bigger gains when the cache size is small. Prefetching on an L2

Table 8 Estimated CPI improvement.

	L1 hit	*L2 hit*	*L2 miss*	*Total*
TSO	0.343	0.049	0.029	0.421
IMS	0.345	0.049	0.064	0.458
CICS	0.379	0.050	0.039	0.468
RAMPC	0.349	0.039	0.097	0.485
FPC1	0.327	0.061	0.150	0.538

Table 9 Prediction using LDT, AGEN, and AGI detection for various L1 cache sizes.

TSO	*TLB miss* (%)	*L1 hit* (%)	*L2 hit* (%)	*L2 miss* (%)	*Total* (%)
64KB data cache					
Correct	0.51	82.40	2.34	1.51	86.76
Incorrect	1.01	8.13	0.55	0.50	10.19
Total predicted	1.52	90.53	2.89	2.01	96.95
No prediction 3.05%					
32KB data cache					
Correct	0.51	81.39	3.37	1.48	86.75
Incorrect	1.01	7.79	0.90	0.50	10.20
Total predicted	1.52	89.18	4.27	1.98	96.95
No prediction 3.05%					
16KB data cache					
Correct	0.51	79.81	4.87	1.52	86.71
Incorrect	1.02	7.31	1.40	0.50	10.23
Total predicted	1.53	87.12	6.27	2.02	96.94
No prediction 3.06%					
8KB data cache					
Correct	0.51	77.55	7.16	1.48	86.70
Incorrect	1.02	6.83	1.88	0.51	10.24
Total predicted	1.53	84.38	9.04	1.99	96.94
No prediction 3.07%					
4KB data cache					
Correct	0.51	74.12	10.51	1.55	86.69
Incorrect	1.02	6.21	2.50	0.51	10.24
Total predicted	1.53	80.33	13.01	2.06	96.93
No prediction 3.08%					

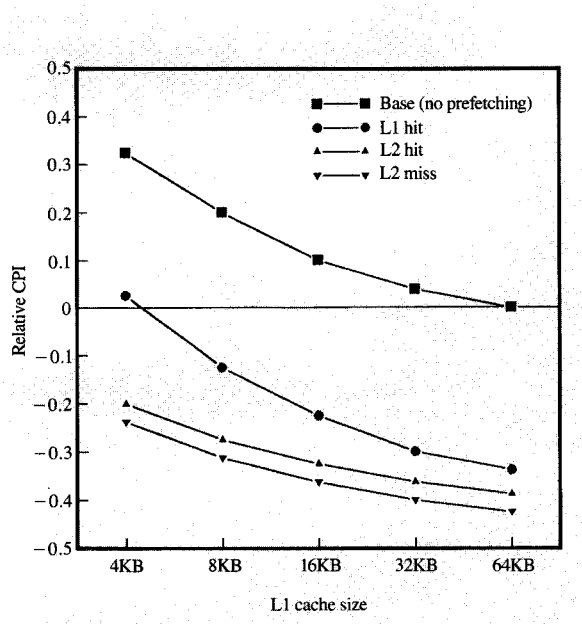

Figure 4

Effect of L1 cache size and prefetch policy.

miss is roughly constant for variations in L1 size, since the L2 size was not varied. The differences appear in the "L1 hit" and "L2 hit" curves. The opportunity to prefetch increases, and the difference between correct and incorrect predictions on a miss also increases. It is clear from the graph that a small cache with prefetching is more effective than a large cache without prefetching. For larger L1 caches, the benefit of prefetching is less, as expected. However, there is always some benefit to prefetching the data on an L1 hit in order to reduce the load instruction to zero-cycle execution.

- *Other performance issues*

Since the load prefetch adds another requestor for cache services, in addition to normal loads and stores, there are performance issues to consider. An additional cache port for prefetching reduces many performance concerns, but may be an unnecessary expense. As long as the cache utilization is low enough, the extra traffic for prefetching may not be significant. A correct prefetch does not increase the cache bandwidth requirements; it only shifts the time of the fetch. For some instructions, no prediction is made, which imposes no additional bandwidth requirements. Incorrect prefetches do increase the workload on the cache, but this applies to only about 10% of load instructions and, considering stores and multiword operations, accounts for about 5% of all data memory references.

On a typical sequential machine, when there is a cache miss, all instructions must wait. When loads can be processed out of order, it is beneficial to allow loads to access the cache while a cache miss from an earlier instruction is in progress. If the subsequent cache access is a hit, it can be processed. If it is a miss, it must wait until the first miss is completed. The load unit accesses the cache when it can find available cache cycles. If there are none available, the prefetch is delayed until it is in sequential order. When a cache miss blocks all other cache access, this may happen enough times to make prefetch less useful, since there may be few available cycles in which to prefetch. If this is the case, a cache that allows other accesses during a miss results in much better performance. Similarly, a miss on a prefetch locks out correct fetches unless the cache can respond while the miss is in progress.

Several issues were not modeled in the performance section. On a first-level cache miss, it is assumed that the requested word is provided first. Additional cycles are required for the remainder of the cache line. This effect can prevent other cache misses from being processed, since the bus is busy. If needed, there are techniques to reduce this penalty. A store queue access to the cache happens on available free cycles, but prefetching also uses some of these cycles. Therefore, the store queue may fill more often, resulting in lost cycles. The priority for the cache should generally be that a full store queue has highest priority, then normal fetches, then prefetches, then stores. Store queue and load queue matches were not counted. A store followed by a load from the same address can result in either the load queue or store queue detecting the address match in the other queue, as discussed in more detail below. Typically, the load queue prefetch must be invalidated, which affects performance. A prefetch to the wrong location, resulting in a cache miss, causes another line to be replaced. Should the replaced line be needed again, another miss will occur (which could be prefetched). This is more significant with lower-associativity (e.g., direct-map) caches. Each of these issues lowers the expected performance gain of prefetches. However, prefetching improves performance very significantly; even with some reduction in this improvement, substantial benefits remain.

Load unit implementation

This section describes hardware that might be used to implement the load unit with prefetching. The implementation described in this section uses an LDT for prediction, together with the AGEN unit and load–load AGI detection. All instructions which fetch data from storage as their first storage operation are handled in the load unit, except for certain complex or privileged instructions, as described previously. Only one operand is

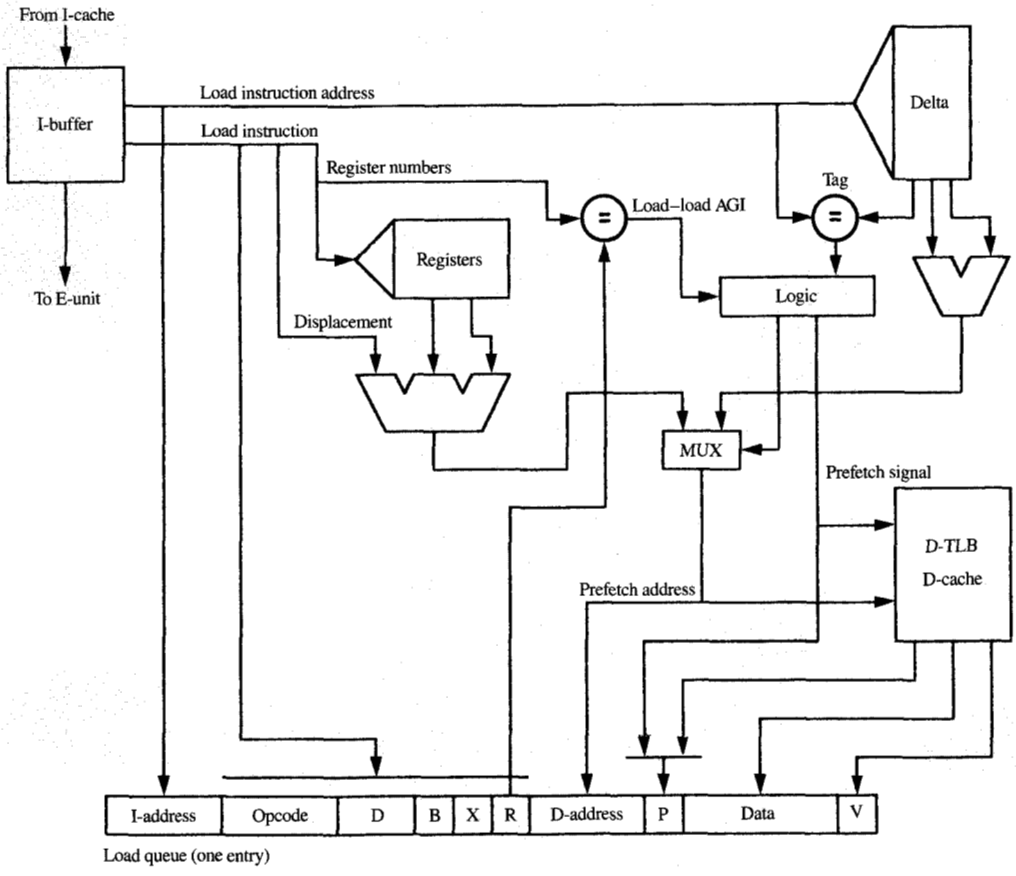

Figure 5

Load prefetch hardware.

prefetched, and only one prefetch is made for the operand. In the ESA/390 architecture, operand 2 is usually a fetch and is the only operand that is prefetched. Should an operand span many sequential bytes of storage, the prefetch of the first bytes prefetches a cache line which effectively prefetches many additional bytes. As described in this section, prefetch includes first-level cache hits and first-level cache misses that hit or miss in the second-level cache. There are subsets of this prefetching strategy which could also be implemented. A TLB miss resulting from a prefetch is not processed from the load unit.

Instructions in the instruction buffer are scanned by the load unit. Typically there are several instructions waiting in the instruction buffer at any time. When a load instruction is found, it is processed by the load unit. If the instruction is a pure load, the required actions are to fetch from storage and write to a register. Since this is typically a one-cycle operation, the instruction is executed entirely in the load unit. The main processor pipeline does not process the load instruction, but it does track the presence of the load. The prefetched data are held in the load queue until the prefetch prediction can be verified; then the register is modified and the load queue entry is removed. For load instructions that require additional actions, such as a computation on the fetched data, storing the data to a different storage location, or further fetches and stores, the remaining execution is handled in the main pipeline. The prefetched data are held until the prediction can be verified; then the data are available for the remainder of the instruction execution. The first fetch does not appear in the main pipeline, resulting in a reduction in the total cycles for the instruction.

Figure 5 shows the hardware for prediction and the formatting of a load queue entry. The load queue consists

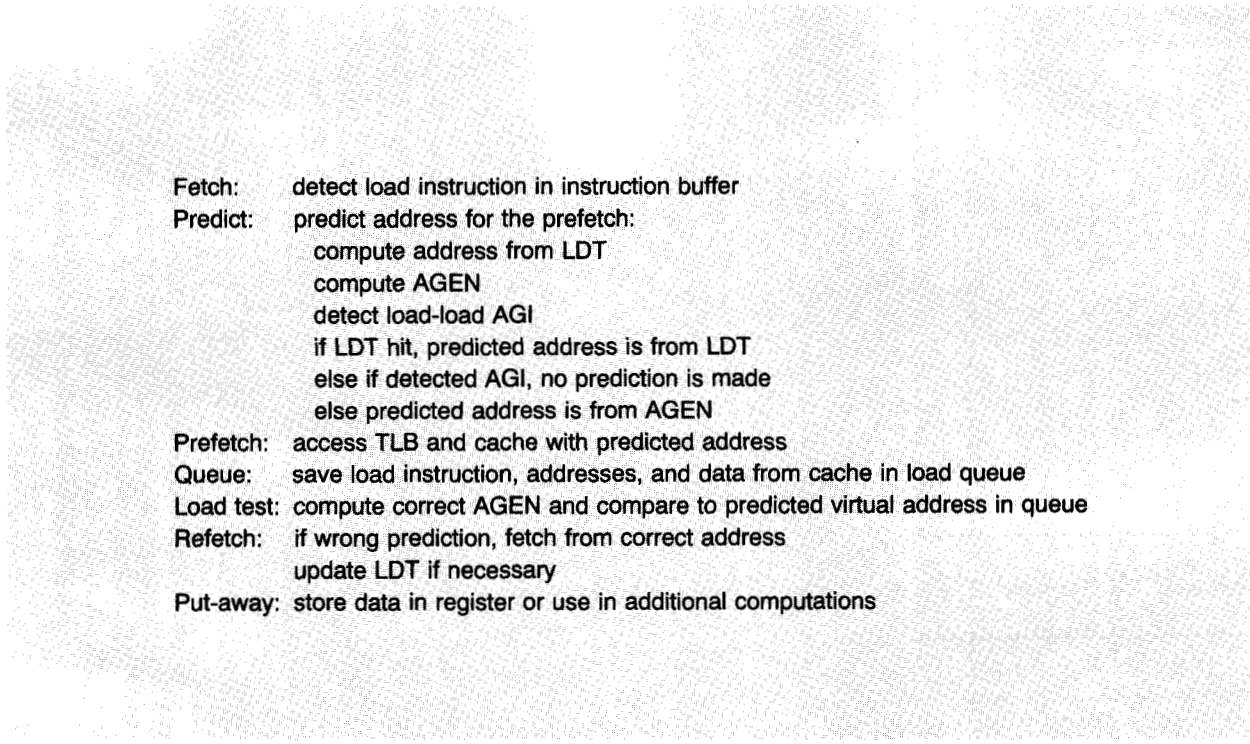

Figure 6

Load unit pipeline description.

of some number of entries determined by performance and cost constraints. The processor execution rate, instruction buffer size, and cache timings determine the optimal size of the load queue. It is expected that approximately four entries is an appropriate size to use for the processor configuration used here. Each entry in the load queue contains the virtual address of the load instruction and the fields of the instruction used in prediction: opcode, base register (B), index register (X), displacement (D), and target register (R). Not all fields are used by all instructions. The B, X, and D fields are zero when not used, as specified in ESA/390 architecture. The R field contains an extra bit to indicate whether or not it is used, since all register numbers are valid. The load queue also contains the virtual address and real address predicted for the prefetch, the data returned from the cache, and two status bits: prefetch (P) and valid (V). When a prefetch is to be made, P is set to one. In cases where no prediction is made, P is zero. In the event of a TLB miss on a prefetch, P is set to zero, canceling the prefetch. The real address is returned from the TLB and is used for data consistency checking. For a TLB miss, no real address is needed. The V bit is set to one when the data are received from the cache.

Figure 6 summarizes the pipeline stages for the load unit shown in Figure 5. When a load has been detected in the instruction buffer, processing begins in the load unit. The address is predicted from the LDT and AGEN. If there is a hit in the LDT, that prediction is used. The registers used to compute the address are compared to modified registers of loads already in the load queue—those loads prefetched but not yet verified. For an LDT miss and a detected load–load AGI, no prefetch is made. If no AGI is detected in this case, the AGEN address is used for prediction. When a prediction is made, the address is sent to the TLB and cache, and the "prefetch" signal is raised. This indicates to the TLB that a miss is not to be processed. Also, if the implementation does not prefetch an L1 or L2 cache miss, the signal is used by the appropriate cache. A prefetch that misses TLB (or a cache, if implemented) returns a "miss" signal to cancel the prefetch in the load queue. When the cache responds with the data, the data are aligned according to the predicted address, stored in the load queue, and marked valid. Note that, if there is time, prediction is improved slightly if unaligned data are stored and the alignment is performed on exit from the load queue using the correct address to control the alignment. There are some cases

where the predicted delta is very close; e.g., if the bus is 64 bits wide and the instruction is predicted to load 32 bits, sometimes the other 32 bits are the correct ones.

A pure load is not processed by the main pipeline; however, there is a bit in the pipeline corresponding to the previous instruction to indicate that a load was present in the instruction stream. Other prefetched loads do appear in the main pipeline. However, the extra pipeline bit is still used. When the load bit in the pipeline reaches the execution cycle, this triggers the load test stage in the load unit. An AGEN is computed for the load and compared to the predicted address. If the prediction is correct, the prefetched data are available in the next cycle for register put-away or use in computations in the load instruction. After a wrong prediction or no prediction, the correct address is used to fetch the correct data. Should the load have a register dependency on the previous instruction, as indicated by the standard register scoreboard or a similar technique, the load test cycle is delayed for one cycle. However, without the load unit, the AGEN and cache access together require two cycles.

Figures 7(a)–(d) show examples of four cases for a "pure" load instruction. On the left are examples where the load instruction uses the main pipeline, as do most instructions. On the right is the timing when a load uses the load unit pipeline. Each example consists of one instruction per line. Each character represents one cycle. The most common case, that of a correct prediction without an AGI on the previous instruction, is shown first. The prefetch occurs earlier than in the left column. Load test is coordinated with execution of the previous instruction. Since the prediction is correct, the data are stored in the put-away cycle. The next instruction executes one cycle earlier than without the load pipeline. When there is an AGI, both main pipeline and load pipeline are delayed, but the load pipeline still saves a cycle. For incorrect predictions, no cycles are saved. The prefetch is early, but another fetch is needed for the correct data. This fetch corresponds to the same cycle as when there is no load pipeline.

Figure 7(e) shows the most common case for an RX-format load instruction. The load pipeline handles the prefetch, which passes the data to the main pipeline to complete the instruction. In the case shown, the data from the fetch are used for one cycle of computation before a register is modified. For other loads requiring more execution cycles, additional execution stages are added in the main pipeline. These stages may fetch from or store to storage. Each load instruction uses both the load pipeline and the main pipeline. The load pipeline operation is identical to that of a pure load instruction. The main pipeline is similar to that of the instruction following a pure load.

Figures 7(f)–(g) show timing examples for a pure load when there is a cache miss on the prefetch. Some or all of the cache miss is overlapped with the main pipeline operation. The first example shows a cache miss on a correct prediction that is completely overlapped, resulting in six fewer cycles. One cycle is from removing the load from the main pipeline, and the other five are the overlapped cache miss penalty. The second example represents a level-two cache miss which requires many cycles. Some overlap occurs, saving five cycles. Had the prediction been incorrect, the L1 miss and L2 hit would have been completely overlapped. There is no saving in this case, but with the load instruction detected sufficiently early, there is no penalty. For an L2 miss, some cache cycles may not be overlapped, resulting in a performance loss. However, this represents the rare worst case which results when both an incorrect prediction and an L2 miss occur. The positive results of prefetching occur much more frequently, resulting in a net increase in performance.

Architectural issues in load unit implementation

Several issues must be addressed in the implementation of the load unit to maintain architectural consistency and good performance. Typical high-performance processors have a store queue. Store instructions create a store queue entry, and the stores are reflected in the cache whenever cache cycles are available. This reduces cache contention between reads and writes and prevents pipeline stalls on store misses. In addition, the store queue helps maintain the required appearance of sequential instruction execution even when instructions are actually executed out of order. Two issues arise when there are both a store queue and a load queue. A load may prefetch from an address that is also in the store queue. Therefore, all load prefetches must search the store queue for matches. The other case is a store to an address that is already in the load queue. Therefore, all stores must search the load queue for matches. In both cases, the correct value for the prefetch is in the store queue and may not yet be in the cache. There are several solutions to these issues. First, the architecture may specify that the store data cannot be passed to the load unit. In ESA/390, for example, in a multiprocessor configuration all processors must observe the store at the same time. The load prefetch violates this rule. When the store is to a load queue entry, the load queue entry could be marked to inhibit prefetch. The fetch would then occur normally and use the standard rules for this situation. When the load is from a store queue entry, prediction could be inhibited, or the load prefetch could be delayed until the store occurred or the actual fetch was reached. In other cases, the architecture might allow the load prefetch to use the data from the store queue, or the store might update the load queue entry.

In many architectures there are some instructions that serialize execution. This typically means that all preceding

```
 Without load pipeline                  With load pipeline          Cycles saved

(a) Correct prediction, no AGI
      F--DAEU                           F--DAEU
L:    F---DARU                          FPTQQLU                          1
      F----DAEU                         F---DAEU
(b) Correct prediction, AGI
      F--DAEU                           F--DAEU
L:    F---D.ARU                         FPTQQQLU                         1
      F-----DAEU                        F---D.AEU
(c) Incorrect prediction, no AGI
      F--DAEU                           F--DAEU
L:    F---DARU                          FPTQQLRU                         0
      F----DAEU                         F---D.AEU
(d) Incorrect prediction, AGI
      F--DAEU                           F--DAEU
L:    F---D.ARU                         FPTQQ Q L RU                     0
      F-----DAEU                        F---D . .AEU
(e) Correct prediction, no AGI
      F--DAEU                           F--DAEU
RX:   F---DAREU                      /  FPTQQLU                          1
                                     \  F---DAEU
      F-----DAEU                        F----DAEU
(f) Correct prediction, no AGI, L1 miss, L2 hit
      F-----DAEU                        F-----DAEU
L:    F------DArrrrrRU                  FPrrrrrTLU                       6
      F-------D.....AEU                 F------DAEU
(g) Correct prediction, no AGI, L1 miss, L2 miss
      F-----DAEU                        F-----DAEU
L:    F------DArrrrrrrrrrrrrrrrrrrrrrRU FPrrrrrrrrrrrrrrrrrrrrrrTLU      5
      F-------D.....................AEU  F------D.................AEU

F   instruction fetch
-   waiting in instruction buffer
D   instruction decode
A   address generation
E   execution (nonload)
R   cache read
r   cache read miss delay
W   cache write
U   register update
P   predict load address
T   target fetch of load data
Q   waiting in load queue
L   load-test
.   pipeline stall
```

Figure 7

Load pipeline timing examples.

instructions must complete before such an instruction can be processed. The simple implementation is to flush all prefetch buffers (instruction buffer, load queue, and any others, such as a branch queue). The instructions after the serializing instruction are then fetched again. The load unit processes them again with the normal prediction, prefetch, and enqueue operations. More complex solutions are to detect which entries must be flushed and which can remain.

Another area of concern with prefetching is that a correct prefetch can occur before the refetch for an instruction that prefetched incorrectly. That is, the correct fetches are performed out of order. This is a concern in a multiprocessor configuration, in that incorrect behavior can result if another processor is storing to the same addresses but prediction results in a different access order. The fetches must appear to be in order from the perspective of another processor. This is important only if the other processor stores to the address of one of the fetches. For caches, there are a variety of algorithms to maintain the consistency of data [25]; these must be extended to the load queue. The straightforward solution is for the load queue to respond to stores from other processors. This may create a large amount of traffic for load queue checking. In caches, a duplicate directory is often maintained for this reason. A duplicate of the load queue (data real addresses only) can be maintained by the cache. To avoid excessive coordination between the cache and load queue, the load queue does not signal the cache when an entry is removed from the load queue. Since the cache copy of the load queue is the same size as the actual load queue, entries are removed from the cache copy when the queue is full. Whenever the cache detects a store matching a cache load queue entry, it can either signal the load queue to change all entries to no prefetch, or it can send the address to the load queue for it to check for an exact match. The cache copy of the load queue is affected by more stores than the real load queue would be, because it reflects stores to addresses which may no longer be present in the real queue, but it filters out most stores, and net traffic to the real load queue is reduced.

Conclusions

A new hardware technique has been presented for improving processor performance. In the proposed mechanism, load instructions are processed early, fetching from a predicted address. This approach provides the potential to make significant performance improvements. By fetching the data early, a cycle is removed from the normal execution of many instructions. The prefetch allows cache misses to be overlapped with other execution, also improving performance. Experimentation strongly suggests that the techniques described in this paper result in approximately 0.45 CPI reduction.

Acknowledgments

The authors thank the anonymous referees for their suggestions, which improved the readability of the paper.

References

1. T. Agerwala and J. Cocke, "High Performance Reduced Instruction-Set Processors," *Research Report RC-12434*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1987.
2. N. P. Jouppi and D. W. Wall, "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines," *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, ACM, Boston, 1989, pp. 272–282.
3. R. R. Oehler and R. D. Groves, "IBM RISC System/6000 Processor Architecture," *IBM J. Res. Develop.* **34**, No. 1, 23–36 (January 1990).
4. R. W. Horst, R. L. Harris, and R. L. Jardine, "Multiple Instruction Issue in the NonStop Cyclone Processor," *Proceedings of the 17th Annual International Symposium on Computer Architecture*, IEEE, Seattle, May 1990, pp. 216–226.
5. G. B. Steven, R. G. Adams, P. A. Findlay, and S. A. Trainis, "iHARP: A Multiple Instruction Issue Processor," *IEE Proc. E, Comput. Digit. Tech.* **139**, No. 5, 439–449 (September 1992).
6. A. J. Smith, "Cache Memories," *Comput. Surv.* **14**, No. 3, 473–530 (September 1982).
7. J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Mateo, CA, 1990.
8. *Alpha Architecture Handbook*, Digital Equipment Corporation, Maynard, MA, February 1992.
9. D. Callahan, K. Kennedy, and A. Porterfield, "Software Prefetching," *Proceedings of ASPLOS IV*, ACM, Santa Clara, CA, April 1991, pp. 40–52.
10. A. C. Klaiber and H. M. Levy, "An Architecture for Software-Controlled Data Prefetching," *Proceedings of the 18th Annual International Symposium on Computer Architecture*, IEEE, Toronto, May 1991, pp. 43–53.
11. D. Kroft, "Lockup-Free Instruction Fetch/Prefetch Cache Organizations," *Proceedings of the 8th Annual International Symposium on Computer Architecture*, IEEE, Minneapolis, 1981, pp. 81–87.
12. G. S. Sohi and M. Franklin, "High-Bandwidth Data Memory Systems for Superscalar Processors," *Proceedings of ASPLOS IV*, ACM, Santa Clara, CA, April 1991, pp. 53–62.
13. T. Chen and J. Baer, "Reducing Memory Latency via Non-blocking and Pre-fetching Caches," *Proceedings of ASPLOS V*, ACM, Boston, October 1992, pp. 51–61.
14. N. P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *Proceedings of the 17th Annual International Symposium on Computer Architecture*, IEEE, Seattle, May 1990, pp. 364–373.
15. J. H. Fu and J. H. Patel, "Data Prefetching in Multiprocessor Vector Cache Memories," *Proceedings of the 18th Annual International Symposium on Computer Architecture*, IEEE, Toronto, May 1991, pp. 54–63.
16. J. Baer and T. Chen, "An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty," *Proceedings of Supercomputer '91*, IEEE, Albuquerque, November 1991, pp. 176–186.

17. J. K. F. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *IEEE Computer* **17**, No. 1, 6–22 (January 1984).
18. B. Blaner, T. L. Jeremiah, and S. Vassiliadis, "A Branch Instruction Processor for SCISM Architectures," *Technical Report TR-01.C437*, IBM Enterprise Systems, Endicott, NY, September 1992.
19. *Enterprise Systems Architecture/390 Principles of Operation*, Order No. SA22-7201-0, 1990; available through IBM branch offices.
20. K. Hua, A. Hunt, L. Liu, J. Peir, D. Pruett, and J. Temple, "Early Resolution of Address Translation in Cache Design," *Proceedings of the International Conference on Computer Design*, IEEE, Cambridge, MA, September 1990, pp. 408–412.
21. W. Wang, J. Baer, and H. M. Levy, "Organization and Performance of a Two-Level Virtual-Real Cache Hierarchy," *Proceedings of the 16th Annual International Symposium on Computer Architecture*, IEEE, Jerusalem, May 1989, pp. 140–148.
22. R. J. Eickemeyer and S. Vassiliadis, "Address Generation Predictors," *Technical Report TR-01.C523*, IBM Enterprise Systems, Endicott, NY, September 1992.
23. R. J. Eickemeyer and S. Vassiliadis, "Load Instruction Processor," *Technical Report TR-01.C568*, IBM Enterprise Systems, Endicott, NY, September 1992.
24. R. J. Eickemeyer, "Address Generation Interlocks in Branch Instructions," *Technical Report TR-01.C388*, IBM Enterprise Systems, Endicott, NY, June 1992.
25. P. Stenström, "A Survey of Cache Coherence Schemes for Multiprocessors," *Computer* **23**, No. 6, 12–24 (June 1990).

Received July 31, 1992; revision received April 3, 1993; accepted for publication May 18, 1993

Richard J. Eickemeyer *IBM Application Business Systems, 3605 Hwy. 52 North, Rochester, Minnesota 55901 (RJE at RCHVMV3, rje@vnet.ibm.com).* Dr. Eickemeyer is an Advisory Engineer in the IBM Rochester Laboratory Hardware Design Center. His current assignment is performance analysis of AS/400® processors. Prior to moving to Rochester, he worked in the IBM Glendale Laboratory, Endicott, New York, on system design and processor performance. Dr. Eickemeyer received the B.S. degree in electrical engineering from Purdue University and the M.S. and Ph.D. degrees from the University of Illinois at Urbana–Champaign. His research interests are computer architecture, parallel processing, and performance analysis. Since joining IBM he has received awards which include two levels of the IBM Publication Achievement Award and two levels of the IBM Invention Achievement Award.

Stamatis Vassiliadis *IBM Enterprise Systems, Mid-Hudson Valley Laboratory, P. O. Box 950, Poughkeepsie, New York 12602 (STAMATIS at GDLVM6, stamatis@vnet.ibm.com).* Dr. Vassiliadis received the Dr.Eng. degree in electronic engineering from the Politècnico di Milano, Milan, Italy, in 1978. He is currently employed at the Mid-Hudson Valley Laboratory, IBM Poughkeepsie, after previously being employed at the Glendale Laboratory, IBM Endicott, New York. His work assignments include the development of new computer organizations and architectures, high-level design and technical leadership in the implementation of new computer systems, and advanced research in a variety of computer-related fields. Previous work included participation in the design of the IBM 9370 Model 60 computer system. Since joining IBM he has received a number of awards, including 15 levels of the IBM Publication Achievement Award, 13 levels of the IBM Invention Achievement Award, and an IBM Outstanding Innovation Award for Engineering/Scientific Hardware Design in 1989. In 1990 he was awarded the most patents in IBM. His research interests include computer architecture, hardware design and functional testing of computer systems, parallel processors, computer arithmetic, EDFI for hardware implementations, neural networks, fuzzy logic and systems, and software engineering. Dr. Vassiliadis has been an Adjunct Professor in the School of Electrical Engineering, College of Engineering, Cornell University, Ithaca, New York, and in the Electrical Engineering Department at the Thomas J. Watson School of Engineering and Applied Science, State University of New York (S.U.N.Y.), Binghamton, New York. He is currently a Visiting Professor in the School of Electrical Engineering, College of Engineering, Cornell University, Ithaca, New York.