Improving Memory Access Performance Using a Code Coalescing Unit

Abstract

High clock frequencies combined with deep pipelining employed by many of the state-of-the-art processors have forced cache hit accesses to be multi-cycle operations. For many programs, untolerated load latencies account for a significant portion of total execution time. In this paper, we present a mechanism called the Code Coalescing Unit (CCU) that can identify and eliminate at run-time several load operations. The multi-cycle load operations are converted to register read operations with zero latency.

Our approach works by using a special buffer called the Store Register Rename Buffer (SRRB), to store the addresses and data for the stores. Subsequent loads for the data are avoided by the CCU by appropriate micro-architectural register renaming. The load operations are converted to register read accesses avoiding costly address translation and cache access which typically takes two or three cycles. The SRRB is used as an extension to the regular register file. The SRRB is similar in structure to a regular store buffer, however the forwarding mechanism is different from forwarding of data using a store buffer. While this technique is applicable for any processor, the relevance is higher for processors in the x86 family because programs compiled for x86 processors, which have small register sets, often exhibit excessive register spills and an abundance of redundant memory accesses.

Experiments on eight x86 traces including Microsoft Word for Windows, Microsoft C++ compiler and two SPEC programs demonstrate that approximately 25%, 34% or 42% of all load operations performed by these applications could have been eliminated using a Code Coalescing Unit in conjunction with a Store Register Rename Buffer of 8, 16 or 32 entries respectively.

Keywords: data forwarding, efficient data access, memory access overhead, register spills, run-time code elimination.

* *This work was supported in part by grants from the National Science Foundation and the Oak Ridge Associated Universities.*

1 Introduction

Modern high performance processors try to issue several instructions per cycle, often out of order, to achieve a high instruction completion rate. To achieve this goal, instruction dependencies must be minimized allowing instructions to flow uninterrupted in the pipeline. For many programs, untolerated load latencies account for a significant portion of total execution time. In this paper, we focus on a technique that converts several load operations into register read operations completely eliminating the load latency.

The cache hit rate of many programs are fairly high with the large on-chip caches which are very common today, however, high clock frequencies combined with deep pipelining employed by these processors have forced cache hit accesses to be multi-cycle operations. Many factors contribute to the latency of a cache hit, such as the time spent in computing the effective address of a load, the required address translation, cache tag checking, and the actual cache access itself. Austin et. al. [2] reported results from studies on impact of load latency on instruction completion rate. Their studies were performed on an aggressive 4-way superscalar processor simulator with a 16k data cache. It was shown that reducing the cache hit latency to one cycle from 2 cycles has a sizable impact on all programs tested. The integer codes were seen to obtain more benefit than the floating point codes. In general it was seen that cache hit latency is very critical in determining processor performance.

Typically many latency hiding techniques focus on tolerating the load latency by executing independent instructions while waiting for the data being loaded. For wide issue processors, it is extremely difficult to find other independent instructions to satisfy all the functional units while the load latency is preventing dependent instructions from being issued. Hiding the load latency requires that the look-ahead window of the processor be large to find other independent operations to execute. A large window means that the processor must do more work to choose ready operations. It will also require the fetcher to sustain a high fetch rate which may require predicting beyond several branches. Therefore, even if the probability of a hit in the level one cache is high, the execution engine may incur a multi-cycle delay to acquire operands for use in computation. This is in stark contrast to register-to-register operations which access its operands directly from the register file.

For a given data memory hierarchy, a good approach to reducing load latencies is through efficient register allocation. If the data is available in a register, load instructions are no longer required to access data. Many architectures including the PowerPC, MIPS, SPARC and Alpha have at least 32 general purpose registers allowing the compiler to perform efficient register allocation, however, processors in the x86 family have only eight architecturally visible general purpose registers.

It is widely known that an insufficient number of architecturally visible registers as in the x86 architecture creates false data dependencies, limits compiler optimizations, increases the amount of register spills and results in several redundant memory accesses. A register is spilled by storing its contents to memory thus freeing the register for use by other instructions. When this variable is needed again, it is loaded from memory into a register (which can trigger the spilling of another variable.) This reduces performance for two reasons: Firstly, the overall code size is increased by the additional memory access instructions; and secondly, the pressure on the memory sub-system is increased by executing additional loads and stores that in turn generate additional address computations and data dependencies. A study conducted by Pierce and Mudge [24] showed that i486 programs have roughly twice the number of memory accesses than that of equivalent MIPS R4000 code. Innovative architectural techniques have helped processors such as the PentiumPro [38] to overcome many of the limitations that result from small architecturally visible register sets. For instance, register renaming with the help of a reorder buffer or other means has helped to eliminate all output and anti-dependencies. However the inability of the compiler to perform optimizations that need architecturally visible temporary registers or the abundance of register spills cannot be overcome by register renaming with a reorder buffer. Many of these loads will hit in the L1 data cache, but if the latency of the cache is 3 cycles, several instructions will be blocked if they are dependent on this data and other independent operations could not be found.

It should be remembered that all memory operations even if cache hits, consume more resources than register-to-register operations. Table 1 illustrates and compares the resources required by register-to-register and memory-to-register instructions with the help of the following instructions.

```
add eax, ebx    ;register to register add
add ebx, [ebx]  ;memory to register add
```

The memory to register add consumes a TLB port, a cache port and an ALU (AGU) for the address calculation, in addition to the resources used by the register-to-register addition. The additional resources (adders, register file ports, and cache ports) used by the memory operations may be centralized due to cost, and thus increases contention with other operations that require these resources.

In this paper, we propose a scheme that eliminates several of the redundant memory access instructions by buffering the data in a mechanism similar to the store buffer and renaming the instructions that use the data to access it from the buffer. Our approach works by using a special buffer called the Store Register Rename Buffer (SRRB), to store the addresses and data for the stores. Subsequent loads for the data are avoided by the proposed Code Coalescing Unit (CCU) by appropriate micro-architectural register renaming. The load

Instr type	Register File	ALU	TLB Port	Cache Port	AGU
reg-to-reg	2 ports	1	0	0	0
mem-to-reg	2 ports	1	1	1	1

Table 1: Resources used by register-to-register and memory-to-register operations.

operations are converted to register read accesses avoiding costly address translation and cache access which typically takes 2 or 3 cycles. The SRRB is thus used as an extension to the regular register file. The SRRB is similar in structure to a regular store buffer, however the data forwarding mechanism is different from forwarding of data using a store buffer. While this technique is applicable for any processor, the relevance is higher for processors in the x86 family.

The Code Coalescing Unit (CCU) can be considered as an extension to the fill unit proposed by Melvin, Shebanow and Patt [26]. Their fill unit groups together micro-operations into larger atomic units and stores them into the Decoded Instruction Cache (DIC). The original fill unit grouped together micro-operations into larger atomic units, but the proposed Code Coalescing Unit goes further by eliminating particular load operations in the process of grouping together micro-operations.

Summarizing the motivation for the proposed scheme:

- Memory access operations even when hits consume more resources than register to register operations.
- L1 cache hit latency is often 2 or 3 cycles and constitutes a limitation on performance
- Extra resources consumed by memory operations (in addition to those used by register to register operations) increase the contention for many of the centralized resources of the machine
- Forwarding through store buffer is as slow as L1 cache hit and consumes as many resources as L1 cache hit.
- Small register sets as in the x86 processors result in an abundance of memory spills and the memory performance will improve significantly if the redundant memory accesses could be converted to register-read operations.

The rest of this paper is organized as follows. In section 2, we describe other research related to the proposed technique. In section 3, the proposed coalescing code unit is presented. Section 4 presents the evaluation of the proposed mechanism and section 5 offers concluding remarks.

2 Related Research

Austin, Pnevmatikos and Sohi [2] proposed a scheme for reducing load latency by fast address calculation. They allow effective address calculation to proceed in parallel with cache access, thereby eliminating the extra cycle required for address generation. Our approach has the advantage that we eliminate the address translation and cache access cycles, and avoid the use of costly TLB and cache resources. They observed 14% to 19% program speedups for integer codes and 6% to 7.5% for floating point codes.

AMD's K5 processor [29] overlaps a portion of effective address computation with cache access. The lower 11 bits of the effective address is computed in the cycle prior to cache access. The entire 32-bit effective address becomes ready just in time for the address tag comparison. However this scheme only reduces the load latency as in [2], whereas our scheme completely eliminates it for several load instructions. Other schemes to improve the speed of address calculation and data loads include work by Golden and Mudge [11], who proposed the use of a load target buffer to predict the effective address of a load early in the pipeline, and Steven [31], who presented a four-stage pipeline to speed up stack accesses. Other non-traditional pipeline organizations to improve address computation and load latency include those in [14] and [12].

Most state-of-the-art processors use a store buffer which holds the data from recent stores and forwards it to future instructions that need the data. The MOB or Memory Ordering Buffer in the PentiumPro [3] [4] [9] [38] can be considered as an example mechanism. However, the latency of the loads that get their data forwarded from the MOB is 3 cycles (same as that of the L1 cache) [39] [40]. In our case, for all loads that get converted to a register read, the load latency is totally eliminated.

Widigen et al [36] of AMD proposed an Operand Prefetch Cache (OPC) to predict data values and to eliminate operand read latency. The OPC is a fully associative content addressable memory structure that holds the instruction address, operand address, operand data, valid bits to indicate validity of each of the three fields, and a count field much like the saturation counters in branch predictors. The OPC predicts the operand memory read results when the instruction address matches the entry in the OPC (i.e. an OPC hit). When a hit occurs, the operand is read from the OPC thus avoiding the memory read latency. The prediction is validated later when the actual values are known. Performance studies showed that a 12 Kbyte OPC can predict approximately a third of the memory reads with more than 95% accuracy. Although this scheme is encouraging, it is essentially a special on-chip cache. A similar scheme was also proposed by Chen and Baer [7] in their LAPC (Look-Ahead Program Counter) data prefetching scheme.

Lozano and Gao [18] proposed a scheme to exploit the short life-span of many variables.

They observed that in dynamically scheduled processors using a reorder buffer, the live-range of several variables may occur entirely within the buffer and then there is no need to retire or commit the values produced by these live ranges to the register file. They proposed a scheme that involves compiler analysis and a simple architectural extension to avoid the useless commits and reduce the number of write ports to the register file. For an architecture with only 4 registers, the performance improvement exceeded 22% for a reorder buffer of size 16 entries (26% for a 32 entry reorder buffer). Our scheme is for data forwarding of memory operations, whereas their scheme is for data forwarding for arithmetic operations.

Sodani and Sohi [30] studied the concept of dynamic instruction reuse, noticing that many groups of instructions are executed repeatedly with the same inputs generating the same results. They buffer the outcome of an instruction in a reuse buffer from where future instructions can access it. By dynamically reusing instruction results, they were able to cut down on the resources required to execute the instructions, and cut down the length of the critical paths of computations. The speedup they observed ranged from none in certain benchmarks to 19% in other benchmarks, for a 32-entry reuse buffer. Recent research on value locality and value prediction [17] [28] [34], also exploit the fact that if a register receives a certain value from a specific instruction, it is likely that the register will receive the same value from a future execution of the same instruction, in the next iteration or in the next call to the function.

The proposed mechanism has similarities to or uses features of Melvin et al's Decoded Instruction Cache [26], and Franklin and Smotherman's Fill Unit [10] [32]. Melvin, Shebanow, and Patt's Decoded Instruction Cache (DIC) [26] creates larger atomic units of work from smaller micro-operations. Micro-operations generated from sequentially fetched instructions are compacted into larger atomic units by a fill-unit and stored in the DIC. Subsequently, these larger atomic units are fetched from the DIC and directly executed. Franklin and Smotherman's fill unit [10] also groups RISC instructions into a DIC for later multiple issue. They also demonstrated the use of a fill unit in a CISC machine which resembled the x86 architecture [32]. The fill unit dynamically collects decoded micro-operations into a DIC. Future code fetches to those locations are satisfied out of this cache, bypassing the decoding logic, and thus improving the performance of the instruction fetch and decode units. Although the proposed CCU has similarities to the fill unit, none of the aforementioned research considered eliminating load operations while performing the fill operations. Another related mechanism is Rotenberg et al's Trace cache [27], which is a structure that caches traces or snapshots of the dynamic instruction stream so that instructions that are otherwise noncontiguous appear contiguous. The instructions in the run-time path plus the associated branch information are stored into the trace cache. If the same trace is encountered again in the course of executing the program, it will be available in the trace cache. Our scheme consists of dynamically eliminating some of the load instructions, and if the new code sequence with several loads eliminated can be captured in a DIC or trace cache, further improvement in performance can be obtained.

3 Proposed Mechanism

In addition to components in state-of-the-art superscalar processors, the proposed scheme illustrated in Fig. 1 uses a Code Coalescing Unit (CCU) and a Store Register Rename Buffer (SRRB) and an optional DIC (Decoded Instruction Cache) [26], a trace cache [27], or a loop buffer [1] [33]. (The number and nature of the functional units can vary from the illustration and there can be multiple numbers of the shown functional units.) As usual, instructions are fetched from the i-cache, decoded and the decoded pool of instructions are sent to the reservation station where they wait until issued to the functional units. In processors such as the PentiumPro, the decoding stage also involves the splitting of the x86 instructions into RISC-style micro-operations. Address calculation instructions are issued to the AGU and once the address is computed, the load micro-operation is issued to the load/store unit. The load/store instructions can access the data in the store buffer (eg: Memory Order Buffer or MOB in the Pentium) or the L1 data cache in most of the cases. Store instructions write the results into the store buffer which get written to the L1-data cache once the stores are ready to be committed.

The proposed scheme calls for the following modifications. All store instructions also write into a special buffer called the SRRB (Store Register Rename Buffer). The SRRB is organized as a content addressable memory structure and each entry consists of two items, the effective address and the operand, much like the store buffer (See Fig. 2). When a load instruction is encountered in the reservation station, its effective address is checked against the addresses present in the SRRB. If the current load's effective address is present in the SRRB, the SRRB returns an identifier which can be used by the load instruction as the source instead of the memory address. The load operation is then avoided by reading the value from the SRRB. The load can be replaced by a move from the corresponding SRRB element to the desired destination register (if the destination is an architecturally visible register). If the destination of the load is not an architectural register, the load can be totally eliminated and the micro-operation that uses the data being loaded is renamed to read the data directly from SRRB. If the SRRB is full when a store operation wants to allocate an entry in SRRB, a replacement will be performed according to LRU, FIFO, or another appropriate replacement policy.

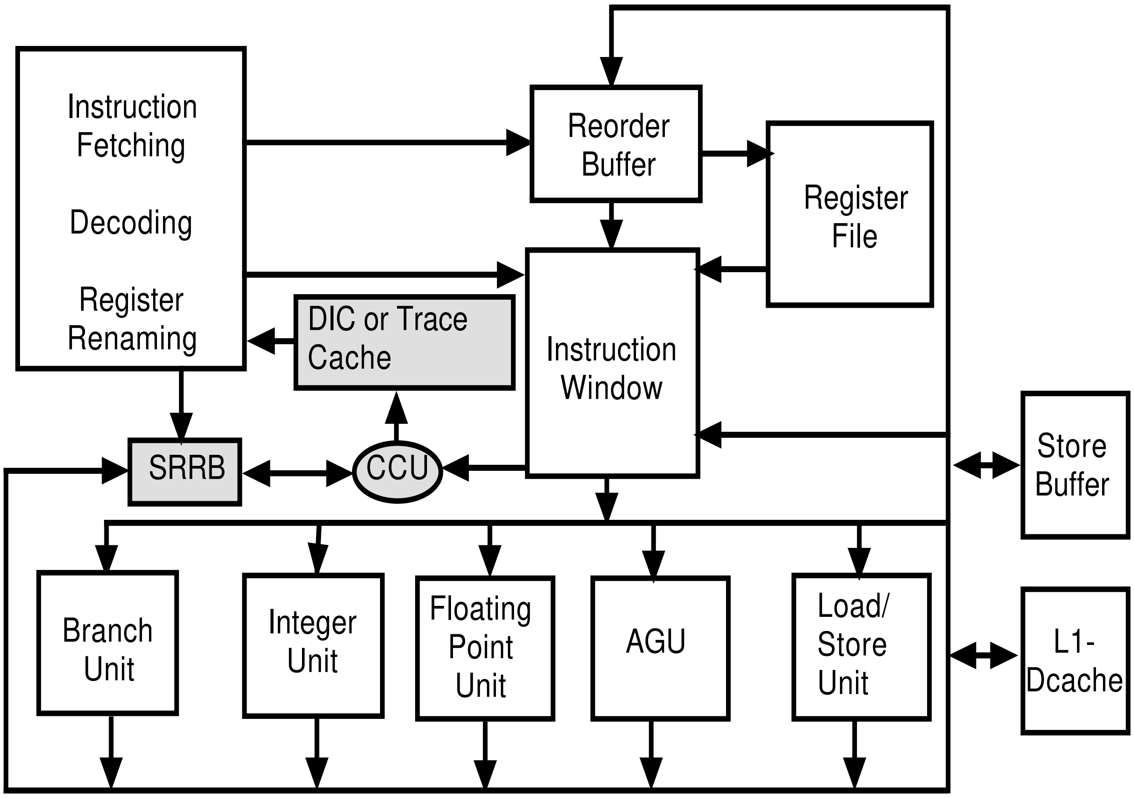

Figure 1: Structure of a processor that incorporates the proposed Code Coalescing Unit

If the processor has a DIC or trace cache, the CCU fills the DIC/trace cache with the modified code, eliminating some of the original load instructions. If a latter instruction fetch address matches that of a DIC entry, the DIC is given priority over the normal instruction cache and decoder path, and decoded micro-operations are supplied directly from the DIC. As a result, the modified (coalesced) code, which has several fewer load instructions, is executed rather than the original code. If a DIC/trace cache is not present the nullification of the load has to be performed each time the load instruction is encountered. In such a case, address computation will be performed each time, but address translation and cache access will still be eliminated. If the modified stream from the DIC or trace cache is executed, address computation, address translation and cache accessing are eliminated. Instead of a DIC or trace cache, a simple loop buffer or instruction buffer similar to the buffers in the IBM 360/91 [1] or CDC 6600 [33], can be employed to capture the modified (coalesced) code.

The proposed scheme can be implemented in a variety of flavors, depending on when exactly the SRRB entry is allocated and whether the SRRB has one entry for an address or whether each instance of an address is a different entry. A few different structures for the SRRB are shown in Fig. 3. In the simplest implementation, which can be used for an

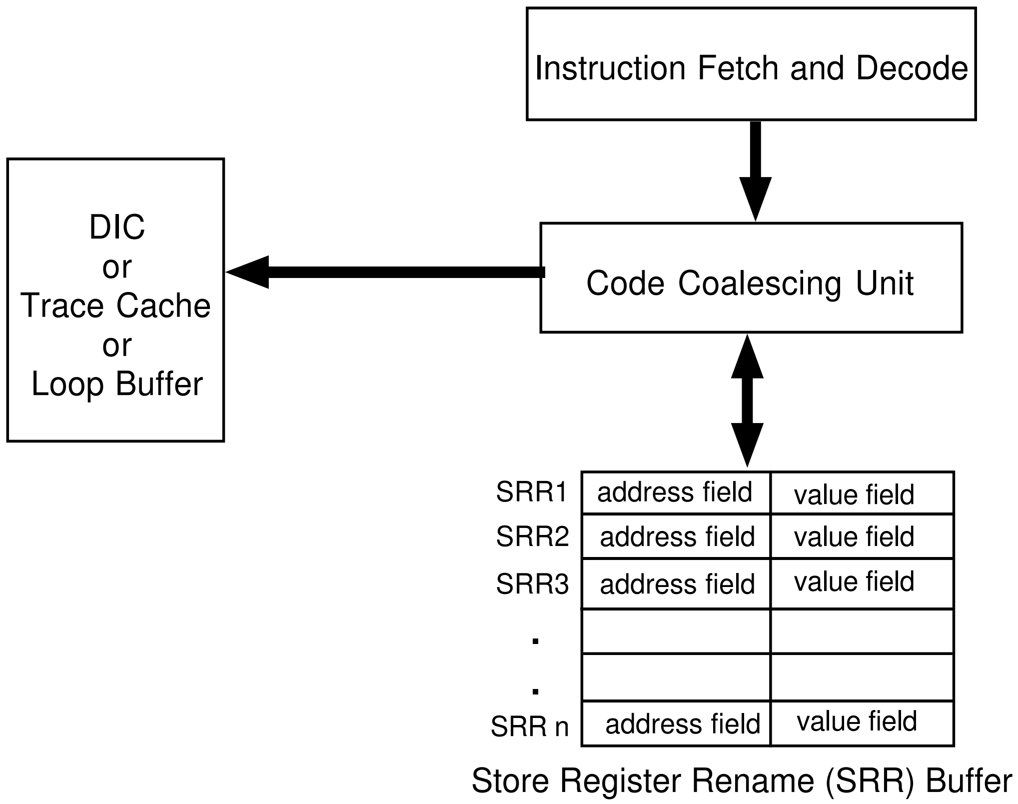

Figure 2: Overview of the Proposed Code Coalescing Unit

in order processor, the store could allocate an entry in the SRRB while executing the store instruction i. e. after the data to be stored is available (See Fig. 3.a). There will be a unique entry for each address and the SRRB could be managed using LRU or FIFO or any other appropriate replacement policy. In an out of order processor, loads and stores may resolve their addresses out of order and a write may 'execute' before its value is known. In such a case, an entry could be allocated in the Store Register Rename Buffer (SRRB) when a store instruction is decoded. If the address or the value to be stored are not available at that point, tags are deposited into the address and value fields of the SRRB and the processor continues. In Fig. 3.c, the tag in the address field points to the AGU instruction that will compute the address for this instruction. The tag in the value field points to the instruction that is computing the result to be stored. As soon as the address generation unit (AGU) computes the address of the store, the address is written into the address field. If the value to be stored is available in a register at the time of decoding, it is copied into the value field at decode time itself. Otherwise, the tag for the value field is replaced by the actual value later when the data value becomes available. The store instructions can get executed (store the value into the SRR buffer) in non-program order. Because each store would create an entry in the SRRB, each instance of an address will be a different entry and the SRRB has to be managed as a FIFO mechanism in order to maintain consistency and perform the renaming correctly. Several store instruction addresses may match a load address and logic must be present to select the youngest matching store. An intermediate scheme would be to allocate an entry in the SRRB after the effective address of the store is computed, but before the value to be stored is available as in Fig. 3.b. Each effective address can have only one entry in the SRRB. If the value to be stored is not available at the point an entry is allocated, tags are deposited into the value field of the SRRB.

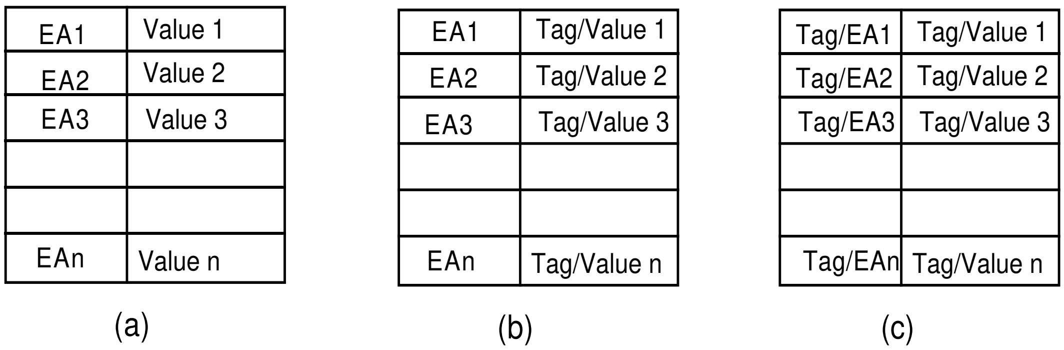

Figure 3: Three possible structures for SRRB. The two fields are the address field and the value field respectively. EA stands for effective address.

If one address can appear in only one entry in the SRRB (as in Fig. 3.a and 3.b), if there is a sequence like

```
store EAX to 2000
load from 2000 to EAX
store EAX to 2000
load from 2000 to EAX
```

the second store instruction cannot allocate an entry in SRRB until the first load is completed, but that is not a major limitation because memory consistency rules do not allow stores to the same address to be performed out of order.

It is necessary to avoid the condition that a load is converted to a register read from SRRB, but the corresponding SRRB entry got replaced before the load actually read the value from the SRRB. If there is no trace cache, this can be implemented as follows. Every entry in SRRB is appended with a counter, which gets incremented by one each time a load is renamed to use the contents and decremented by one each time an actual load uses the entry. An entry from SRRB may not be replaced from the buffer unless the count is zero i. e. there is no pending load which will be expecting to find this data in the SRRB. This will not work in processors with a DIC/trace cache because the translation of the load into a read of SRRB will not occur for each instance of the load once the trace cache has captured the modified instruction stream. If a trace cache is used, the coalesced code will be written into the trace cache. The SRRB should retain the particular entry until the coalesced code is executed. Hence when an entry in the SRRB is being replaced, all coalesced trace cache entries utilizing this SRRB entry should be invalidated. In order to selectively invalidate these cache lines, each cache line must contain the information to identify the SRRB entries it is dependent on, or each SRRB entry must contain information on all trace cache blocks that reference to this entry.

As mentioned before, in a processor such as the PentiumPro, the original x86 instructions are first translated into RISC-style micro-operations. Consider an instruction such as

```
imull  %eax, 12036(%ebp, %edx, 4)  ; multiply instruction
```

This instruction will be split into the following 3 micro-operations. (For sake of simplicity, the use of segment registers and address scaling has not been shown in the list of micro-operations.) The sources and destinations of these RISC-style operations (ROPs or uops) are generally renamed to a larger set of physical registers (for instance, 40 in the PentiumPro), and forwarded to the reservation station and the reorder buffer. In the RISC-style code, the temporary registers or microarchitectural registers are represented as $MARi$.

```
a  MAR1 <- EBP + EDX -12036  ;address computation
b  MAR2 <- memory[MAR4]      ;load operation
c  EAX <- MAR2 * EAX         ;actual multiplication
```

Essentially what is being proposed is a mechanism that combines ROPs from several original instructions and in the process of combining, it eliminates some load operations which are loading spilled data from prior spill stores. The current proposal is to eliminate only the load instructions. Even some of the store instructions could be eliminated, however, that will necessitate schemes not to violate the memory consistency models supported by the processor. As an advanced idea, instead of converting a load into read of a known value, it could be converted into a reference to whatever operation generated the data for the store. This of course involves other issues and will not be dealt with in this paper.

The proposed mechanism can be extremely useful for processors such as x86 with a small architecturally visible register set. Consider a loop with inter-iteration dependencies as in the following example.

```
for (i=0; i<n; i++)
.
.
```

```
x[i]=z[i]*(y[i]-x[i-1])
.
.
```

The x86 assembler code generated for this loop by gcc on a machine with Linux operating system is shown below.

```
movl   %eax, -8024(%ebp,%edx, 4)    ; load y(i) to EAX
subl   %eax, (%ecx)                 ; y(i) - x(i-1)
imull  %eax, 12036(%ebp, %edx, 4)   ; z(i) * (y(i) - x(i-1))
movl   -4012(%ebp, %edx, 4), %eax   ; store x(i)
```

These 4 instructions will be first decoded and split into micro-operations which look similar to **sequence 1** shown below. Six microarchitectural registers have been used here for storing temporary values. (Micro-operation 2.a also involves the addition of ECX with the data segment register, though not explicitly indicated.) In step 10, EAX contains $x(i)$ however due to the lack of registers, in step 2 (in the next iteration), EAX has to be used for $y(i)$. Hence $x(i-1)$ has to be brought from memory in step 4 in the next iteration.

```
1    1.a  MAR1 <- EBP + EDX -8024   ;addr of y(i)
2    1.b  EAX <- memory[MAR1]       ;load y(i)

3    2.a  MAR2 <- [ECX]             ;address of x(i-1)
4    2.b  MAR3 <- memory[MAR2]      ;spill-load x(i-1)
5    2.c  EAX <- EAX - MAR3         ;y(i)-x(i-1)

6    3.a  MAR4 <- EBP + EDX -12036  ;addr of z(i)
7    3.b  MAR5 <- memory[MAR4]      ;load z(i)
8    3.c  EAX <- MAR5 * EAX         ;z(i)*(y(i)-x(i-1))

9    4.a  MAR6 <- EBP + EDX -4012   ;addr of x(i)
10   4.b  memory[MAR6] <- EAX       ;store x(i)
```

Sequence 1 - Original micro-operation sequence

With the presence of CCU, the above original code sequence can get replaced by **sequence 2** shown below. The micro-operation $2.b$ in the original sequence gets eliminated because the micro-operation $4.b$ (in the modified sequence) buffers the data into SRRB. With a trace cache, even micro-operation $2.a$ will be eliminated after the first load, however, if no trace cache is present, only after the address computation does it become known that the load could be eliminated. Micro-operation $2.b$ is eliminated in either case.

```
1.a  MAR1 <- EBP + EDX -8024    ;addr of y(i)
1.b  EAX <- memory[MAR1]        ;load y(i)

2.a  MAR2 <- [ECX]              ;address of x(i-1)
2.c  EAX <- EAX - SRR1          ;y(i)-x(i-1)

3.a  MAR4 <- EBP + EDX -12036   ;addr of z(i)
3.b  MAR5 <- memory[MAR4]       ;load z(i)
3.c  EAX <- MAR5 * EAX          ;z(i)*(y(i)-x(i-1))

4.a  MAR6 <- EBP + EDX -4012    ;addr of x(i)
4.b  SRR1,memory[MAR6] <- EAX   ;store x(i) into memory and SRR1
```

Sequence 2 - Modified micro-operation sequence

Assume that there is a sequence such as the following.

```
movl  (%eax),%eax    ;store
movl  %eax, (%eax)   ;load dependent on previous store
movl  (%eax),%eax    ;store dependent on previous load
movl  %eax, (%eax)   ;load dependent on previous store
```

When this sequence is executed on PentiumPro, load store pairs execute once per 3 cycles [39] and the total execution time is 6 cycles. However with the proposed CCU, load store pairs execute once per 2 cycles, for a total execution time of 4 cycles for the sequence. Fig. 4 illustrates the execution of this sequence with and without the proposed CCU.

In an example like

```
for (i=0; i<n; i++)
x(i) = y(i) + z(i)*sqrt(c)
```

where the compiler wants to optimize the code to

```
d=sqrt(c);
for (i=0; i<n; i++)
x(i) = y(i) + z(i)*d
```

to avoid the computation of $sqrt(c)$ inside each iteration, if all the architectural registers are needed for other computations within the loop, usually the compiler will have to spill the value of $sqrt(c)$ to a memory location, and load it in every iteration. If the compiler

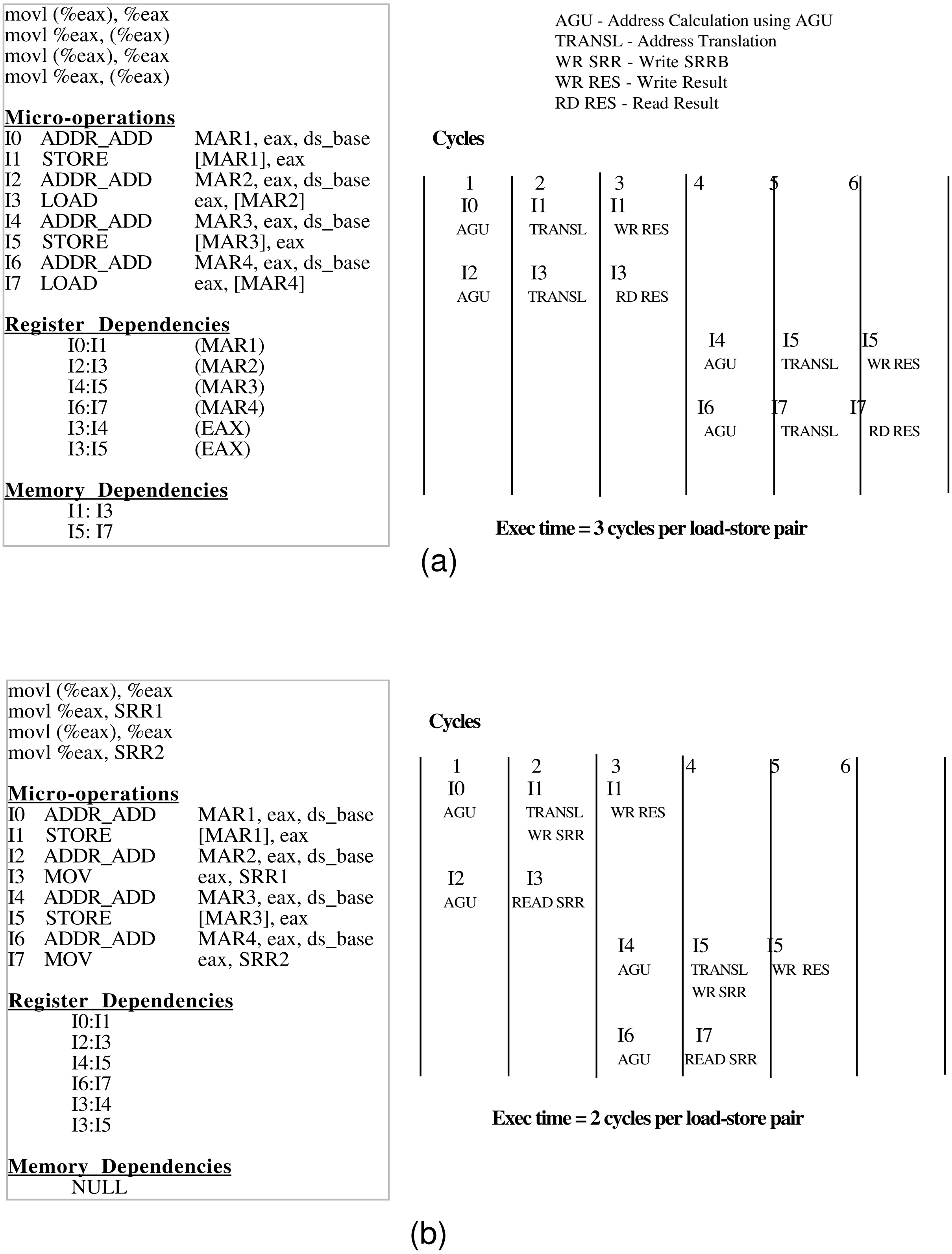

Figure 4: Illustration of the code sequence, micro-operation sequence, register dependencies, memory dependencies, and pipeline execution cycles for an example sequence (a) without CCU (b) with CCU

generates a loop where spill stores are followed by spill loads in every iteration, the CCU can easily identify them and rename them. However, if there is one store before entering the loop and repeated loads in each iteration, and if FIFO replacement policy is used instead of LRU, if there is a sequence such as

```
      store a
L1:   .
      .
      load a
      .
      .
      store y(i)
      loop L1
```

depending on the size of SRRB, at some point, the entry for a will be replaced from the SRRB.

The proposed scheme can work without any assistance from a compiler but compiler assistance can improve the effectiveness of the scheme. Without compiler assistance, it will be the responsibility of the hardware to detect that the effective address of a load is the same as the effective address of a prior store whose value is still available in the SRRB. A compiler can leave a hint whenever a register is spilled which the CCU can use. To accomplish this, there should be 2 kinds of store instructions and 2 kinds of load instructions. One type consists of the normal load and store instruction, and the other type consists of a *spill-load* and *spill-store* instruction. Anytime a temporary variable would have been saved in a register for possible reuse (if there were more registers), the compiler generates a *spill-store* instead of a normal store. If all spill-stores could be accurately marked, only spill-stores need to be saved in the SRRB. This can improve the utilization of the SRRB. Notice that this is only an optimization to improve the efficiency of the scheme and otherwise no compiler help is required for the proper operation of the CCU structure.

A good use of *spill-loads* is in the case of statically allocated compiler variables. Such variables may not generate any store at all. All that the dynamic sequence has is repeated loads of the same value from the memory as in the following sequence. In this case, CCU will not be able to exploit data reuse without compiler help. In this example sequence, a is bound to a value statically and the loop involves repeated loading of this value without any stores to the address.

```
L1 :  .
      .
      load a
      .
      .
      store y(i)
      loop L1
```

If the compiler could generate a *spill load* instead of the normal load, we could modify the mechanism to allocate SRRB entries for spill loads also in addition to stores, and then this kind of data reuse can also be exploited by the CCU. In our performance evaluation in the following sections, we have not incorporated this mechanism and hence the indicated performance improvement does not reflect this optimization.

3.1 Other Issues:

Will this scheme have any applicability for processors other than x86?

Yes. Although the impact of the scheme will be greatest for x86 processors which have high register pressure, the scheme will be valid even for processors with 32 registers. Primarily there are two reasons for this.

(i) Compilers, many time resort to 'aliases' when references cannot be completely resolved at compile time. A compiler cannot always insure that an assignment to a variable (say, a parameter passed to a procedure) does not also modify another variable in the program (say, a global variable also accessed by the procedure) [16]. When variables are aliased, the compiler may generate frequent loads of values recently stored. This can happen even in RISC processors with large register sets and the CCU will be able to exploit the value locality.

(ii) Even in processors with 32 or more general purpose registers, compiler conventions partition the register set into groups for procedure arguments, temporaries preserved across calls, temporaries not preserved across calls, reserved for OS kernels, stack and frame pointers, etc. The 8 or 10 registers allocated for temporaries may not be sufficient in certain programs and can induce spills. Often times, when you analyze register usage in programs (say, using pixstats on a MIPS machine), it may be seen that a small subset (say, 8) of the 32 registers is heavily used and some registers are never used. Hence register spills can be expected to happen even in processors with 32 registers even though their frequency of occurrence will be far less than the equivalent in processors such as the x86.

How's the proposed mechanism different from data forwarding using a store buffer such as PentiumPro's MOB?

Forwarding through MOB takes 3 cycles, just like a hit in the L1 cache. Assuming the presence of a trace cache/DIC, the modified sequence has a read from SRRB operation instead of a load instruction, saving the address calculation, address translation and the MOB access. Even if there is no trace cache or even a loop buffer, the address translation and cache access cycles are saved. Store buffers typically use physical addressing, but it is easy to use virtual addressing for the SRRB and save the address translation step. The synonym problem that arises when virtual addressing is used can be solved in the following manner. Synonym problem arises when two virtual addresses are mapped to the same physical address. Assume that virtual addresses A and C are mapped to the same physical address B. If there is a `store A` followed by a `store C` followed by a `load A`, the data stored to C is what needs to be forwarded by SRRB to the load instruction. When virtual addressing is used for SRRB, incorrect operation can occur if the data from `store A` is forwarded. This can be avoided by invalidating the SRRB entry for A when store to C occurs. An Inverse Translation Buffer can be used or for simplicity, an algorithm such as the following can be used.

```
if
((virtual tag(store0) != virtual tag(store1))
and
(page offset(store0) == page offset(store1)))
invalidate SRRB entry for store0
```

This results in `load A` having to go to the normal store buffer or cache to find the data, but such 'synonyms' occur extremely rarely and the loss in performance will not be significant. Modern compilers normally misalign different data structures and it is extremely unlikely that some coexisting useful data have same page offsets to get flushed due to this procedure.

If a speculative store wrote into the SRRB and a branch mispredict exception occurs, how would the respective entries in SRRB be nullified?

If the SRRB is operated in a FIFO fashion, it is possible to selectively invalidate those entries of the SRRB that were due to speculative stores. Otherwise, the SRRB will be flushed and all trace cache entries with coalesced code will be invalidated (if a trace cache is present).

What happens to SRRB for context switches?

The SRRB can be flushed during context switches without significant loss of performance. Or, the SRRB can be saved when the process is switched out and is restored when the process is switched back in.

4 Performance Evaluation

4.1 Traces and Tools

Our performance analysis is based on trace driven simulation using a set of traces obtained from the Etch project [5]. Etch is an instrumentation and optimization tool for Win32 executables. These traces are gathered under Windows NT 3.51 with Etch v1.36.

Table 2 gives a summary of the traces obtained from the Etch project [5]. It may be observed that there are traces from general purpose personal computer applications such as Microsoft Word as well as scientific applications such as the SPEC. One of the traces come from the `Hello World` program implemented in C and was used only for debugging. Excluding that, there are six traces from integer applications and two traces from floating point applications from the SPEC95 suite. A utility program for parsing traces called *etstat* is available from the Etch web site. This tool decodes binary traces and provides the statistics of different types of instructions. Table 3 illustrates the total number of instructions, the percentage of loads, and the percentage of stores for each trace. As observed the trace sizes range from 4 million to 50 million. It may also be observed that more than half of all instructions in the instruction stream are memory access instructions.

4.2 Memory reference pattern analysis

We performed some profiling of the traces in order to understand the memory reference patterns in the applications. One objective was to obtain the distribution of the stores, regarding the number of loads that follow from that address before another store to that address. Fig. 5 presents a histogram of the stores that have a certain number of loads after the store. Approximately 70% to 80% of all stores are followed by at least one load to that address. Some stores are followed by tens or hundreds of loads to the same address. More than 40% of all stores are followed by exactly one load to that address. This profiling analysis indicates that CCU can potentially eliminate a significant number of loads.

Another interesting aspect quantified was the percentage of all load instructions that appear as loads to an address between stores to the same address. This information is presented in the first data column in Table 4 and in Fig. 6. It is observed that 70% to 95% of all load instructions appear as loads between stores to an address. This statistic also indicates the potential benefit of the proposed scheme. Our profiler also obtained the maximum number of the loads-between-stores for an effective address, which appears in the second column in Table 4. This number ranges from 200 to 20,000 in the various benchmarks.

Trace	Description
Integer Traces	
Perl	This trace is gathered by running a perl program, weblint.pl on a selection of html files. The Perl version used here is Perl for NT Revision 4.0.1.8, patch level 36 by Larry Wall.
MPEGPLY	This is a trace of a Win32 MPEG implementation displaying a video clip of a rocket launch. The trace was generated with MPEGPLY V1.65 by Michael Simmons.
Borland/Intel C++ Compiler (bcc32i)	This is a trace of Intel C++ compiler compiling and optimizing etstat.c, from the Etch Trace utilities.
Microsoft Visual C 4.0 C ++ Compiler (msvc.c1) (msvc.c2)	This is a trace of the Microsoft C v4.0 compiler compiling and optimizing etstat.c, from the Etch Trace Utilities. The two compiler passes included here are c1.exe and c2.exe.
Microsoft Word for Windows (winword)	This is a trace of the Word for Windows version 7, creating a brief document. The first 11560655 bytes of the uncompressed trace are Word startup and initialization; the rest is creation of wordtest.doc.
Hello World (HW)	This is a trace of the *Hello World* program implemented in C. It is a very small program and useful only for debugging.
Floating Point Traces	
su2cor	This is a trace of the first 50 million instructions of su2cor, a Fortran benchmark from SPEC95 suite, generated on a Windows NT 4.0 system.
Tomcatv	This is the trace of the first 50 million instructions of tomcatv, a mesh generation Fortran program from the SPEC95 suite, generated on a Windows NT 4.0 system.

Table 2: A summary of the Etch traces used in our study

Traces	Total Instr	% Loads	% Stores	% Loads+Stores
Perl	19,536,097	36.69	19.42	56.11
MPEGPLY	38,254,807	26.98	16.85	43.83
Borland/Intel C++ Compiler	20,202,721	31.15	18.30	49.45
MS C Compiler c1.exe	4,158,086	38.57	22.49	61.06
MS C Compiler c2.exe	7,240,678	31.09	17.61	48.70
MS Word for Windows	10,924,853	45.21	28.57	73.78
su2cor	50,000,000	40.59	28.63	69.22
tomcatv	50,000,000	29.46	22.61	52.07

Table 3: Load-Store Mix of the Traces

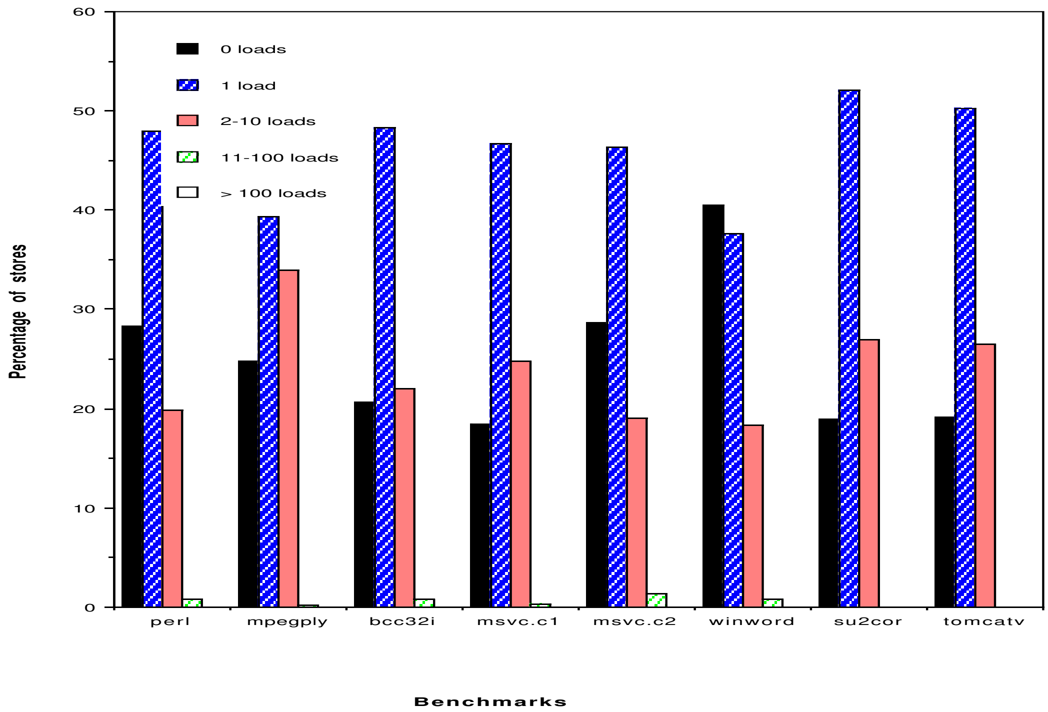

Figure 5: Analysis of stores - Percentage of stores followed by a certain number of loads

Traces	% loads between stores	Max loads btn stores
Perl	71.86	4763
MPEGPLY	95.35	20451
Borland/Intel C++ Compiler	80.60	9687
MS C Compiler c1.exe	69.39	2093
MS C Compiler c2.exe	79.64	1945
MS Word for Windows	73.10	19554
su2cor	92.59	1007
tomcatv	95.03	260

Table 4: Loads to an address appearing between stores to the same address

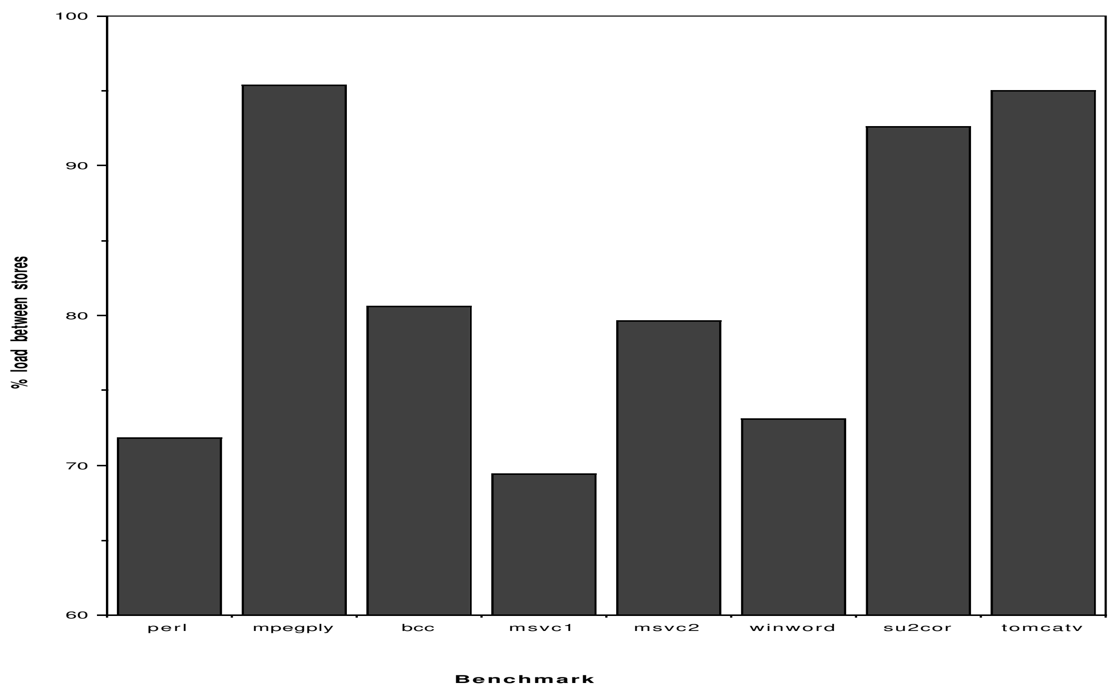

Figure 6: Percentage of load instructions that appear as loads between stores

Benchmark	1 entry	2 entries	4 entries	8 entries	16 entries	32 entries
perl	5.57	10.70	17.49	27.00	34.93	42.55
mpegply	3.63	6.93	16.82	34.01	44.95	47.25
bcc32i	7.49	12.96	20.67	31.46	42.40	52.72
msvc.c1	6.12	12.76	18.58	25.77	38.75	46.46
msvc.c2	5.57	9.23	16.33	25.01	34.71	44.32
winword	5.46	9.02	14.57	22.69	30.66	39.26
su2cor	3.90	10.55	16.74	37.27	43.22	44.17
tomcatv	1.77	1.78	1.79	9.73	17.35	28.75
Geo Mean	4.58	8.13	12.98	25.01	34.61	42.62

Table 5: Percentage of loads that can be eliminated by coalescing

4.3 Loads that can be eliminated

We performed simulations assuming store register rename buffers of various sizes up to 32 entries to evaluate the potential number of load instructions that can be eliminated if a Code Coalescing Unit is employed. The results are presented in Table 5 and Fig. 7. Experiments were performed with LRU and FIFO replacement policies. The information in Table 5 is based on a Store Register Rename Buffer operated with the LRU replacement policy. It is observed that approximately 25%, 34% or 42% of all load instructions in the benchmark programs could be eliminated using a Store Register Rename Buffer of 8, 16 or 32 entries respectively.

Fig. 8 illustrates a comparison of performance for FIFO replacement policy in comparison to LRU. It is observed that FIFO replacement policy performs almost as good as LRU. In general, LRU is better than FIFO however there are one or two anomalous cases with FIFO performing better than LRU (eg: tomcatv for SRRB=16 entries).

5 Summary and Concluding Remarks

In this paper, we presented a mechanism called the Code Coalescing Unit (CCU) that can identify and eliminate at run-time several load operations. We make use of previously proven structures, such as the DIC, the fill unit, and the trace cache and incorporate a special buffer to store the data from register spilling stores and forward it to subsequent loads through an appropriate micro-architectural register renaming scheme. The mechanisms such as fill unit, DIC and trace cache are meant for improving instruction fetching or decoding for multiple issue. The CCU is not for improving instruction fetching, but it is for improving data fetching. The CCU technique is orthogonal to mechanisms such as the DIC, trace cache and

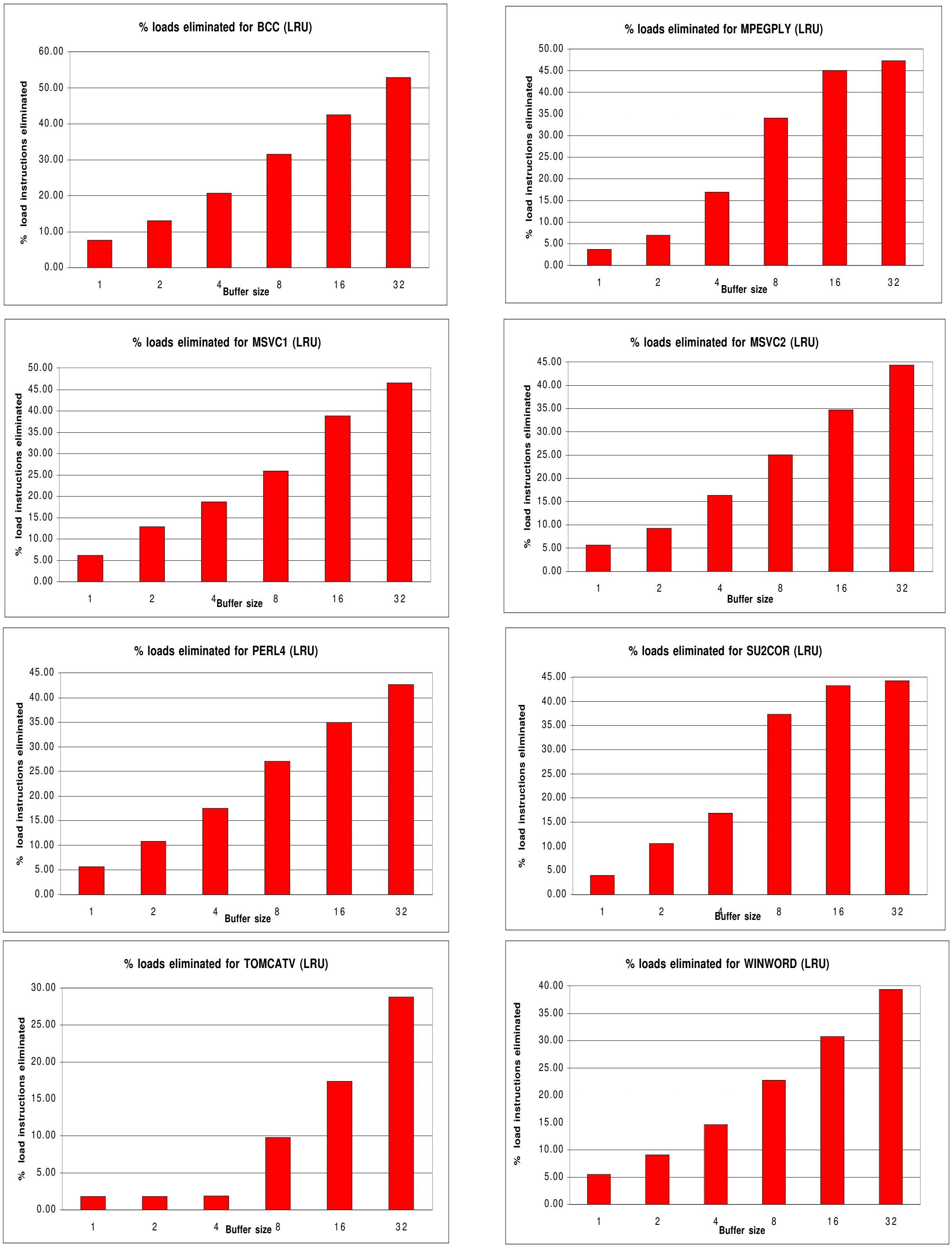

Figure 7: Percentage of load operations that can be eliminated by coalescing

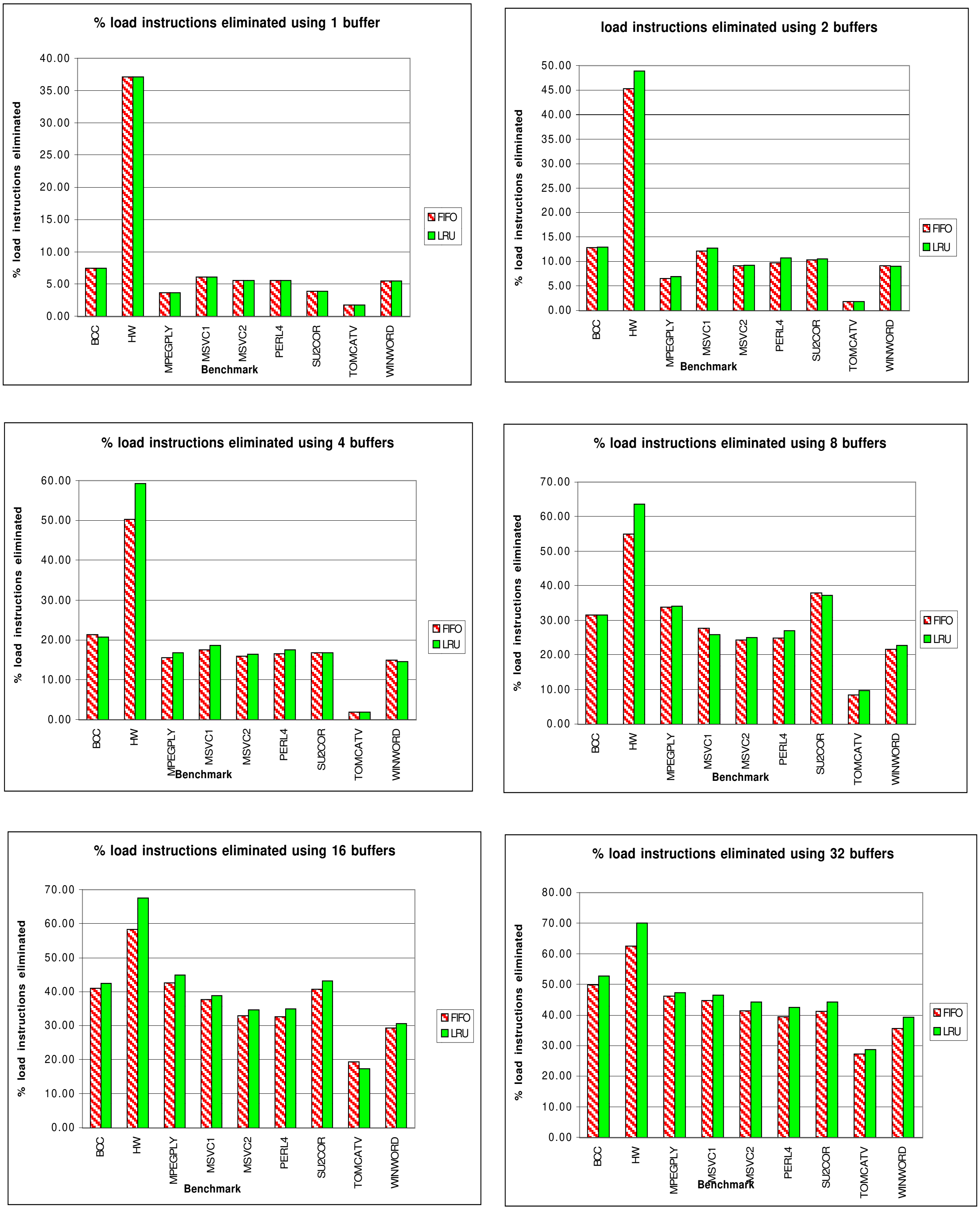

Figure 8: Percentage of load operations eliminated (LRU vs FIFO)

fill unit. The instruction fetching and decoding advantages from trace cache, DIC or fill unit will continue to be present. Mechanisms such as the trace cache, and DIC actually make it possible to obtain the full benefits of CCU. Partial benefits can be obtained without the DIC/trace cache (i. e. with only the CCU and SRRB). The CCU reduces the net amount of work to be performed by a program in order to accomplish a task. This can lead to savings in signal activity and power.

While this technique is applicable for any processor, the relevance is higher for processors in the x86 family which have high register pressure. Several different implementations of this idea suitable for different processor microarchitectures can be devised with appropriate modifications. Important details of the scheme for in order and out of order processors have been presented. Experiments on seven x86 traces including Microsoft Word for Windows, Microsoft C++ compiler and two floating point programs from the SPEC95 suite demonstrate that approximately 25%, 34% or 42% of all load operations performed by these applications could have been eliminated using a Code Coalescing Unit in conjunction with a Store Register Rename Buffer (SRRB) of 8, 16 or 32 entries respectively.

References

[1] D. W. Anderson, F. J. Sparacio, R. M. Tomasulo, "Machine Philosophy and Instruction Handling", IBM Journal of Research and Development, 11, No.1, pp. 8-24, January 1967.

[2] T. M. Austin, D. N. Pnevmatikatos, and G. S. Sohi, "Streamlining Data Cache Access with Fast Address Calculation," Proceedings 22nd International Symposium on Computer Architecture, June 22-24, 1995, pp. 369-380.

[3] D. Bhandarkar, "RISC versus CISC: A Tale of Two Chips", Computer Architecture News, Vol. 25, No. 1, March 1997

[4] D. Bhandarkar and J. Ding, "Performance Characterization of the PentiumPro Processor", Proceedings of the Third High Performance Computer Architecture Symposium, 1997, pp. 288-297.

[5] The Etch web sites in Harvard University and University of Washington, Seattle. Traces from "http://etch.eecs.harvard.edu/traces/"

[6] B. Calder, P. Feller and A. Eustace, "Value Profiling", Proceedings of MICRO-30.

[7] T-F Chen, J-L Baer, "Effective hardware-based data prefetching for high performance processors", IEEE Transactions on Computers, Vol. 44, No. 5, pp. 609-623.

[8] T. Conte, K. Menezes, P. Mills and B. Patel, "Optimization of instruction fetch mechanisms for high issue rates", 22nd International Symposium on Computer Architecture, pp. 333-344, June 1995.

[9] R. P. Colwell and R. L. Steck, "A 0.6μm BiCMOS Processor with Dynamic Execution", Proceedings of ISSCC, February 1995, pp. 176-177.

[10] M. Franklin and M. Smotherman, "A Fill-Unit Approach to Multiple Instruction Issue", Proceedings of Micro-27, San Jose, 1994, pp. 162-171.

[11] Michael Golden and Trevor Mudge, "Hardware Support for Hiding Cache Latency", U. of Michigan, EECS Dept., CSE-TR-152-93.

[12] Michael Golden and Trevor Mudge, "A Comparison of Two Pipeline Organizations", Proceedings of the 27th Annual International Symposium on Microarchitecture, November 1994, pp. 153-161.

[13] W.M. Hwu and Y.N. Patt, " HPSm, A High Performance Restricted Data Flow Architecture Having Minimal Functionality," Proc. ISCA, Tokyo, 1986, pp. 297-306

[14] N. M. Jouppi, "Architecture and Organizational Tradeoffs in the Design of the MultiTitan CPU", Proceedings of the 16th Annual International Symposium on Computer Architecture, May 1989, pp. 281-289.

[15] Author list suppressed for anonymity, "Improving CISC Memory Access Performance Using a Code Coalescing Unit", Technical Report, TR 97-0620- Affiliation suppressed for anonymity.

[16] M. Johnson, "Superscalar Microprocessor Design", Prentice Hall, 1990.

[17] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value Locality and Load Value Prediction", Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII), October 1996.

[18] L. A. Lozano C., and G. R. Gao, "Exploiting Short-Lived Variables in Superscalar Processors", Proceedings of MICRO-28, 1995, pp. 292-302.

[19] G. Tyson and T. Austin, "Improving the Accuracy and Performance of Memory Communication Through Renaming", Proceedings of MICRO-30.

[20] Y.N. Patt, M.W. Hwu, and M.C. Shebanow, "HPS a New Microarchitecture: Rationale and Introduction," Proc. Micro-18, Asilomar, December 1985, pp. 130-108.

[21] Y.N. Patt, S.W. Melvin, W.M. Hwu, M.C. Shebanow, C. Chen, and J. We, "Run- Time Generation of HPS Microinstructions From VAX Instruction Stream," Proc. Micro- 19, New York, December 1987, pp. 162-167.

[22] Y.N. Patt, S.W. Melvin, W.M. Hwu, and M.C. Shebanow, "Critical Issues Regarding HPS, A High Performance Microarchitecture," Proc. Micro-18, Asilomar, December 1985, pp. 109-116.

[23] Andrew R. Pleszkun, "Techniques for Compressing Program Address Traces," MICRO-27, 11/94, pp.32-39.

[24] J. Pierce and T. Mudge, "IDtrace - A Tracing Tool for i486 Simulation", Technical report CSE-TR-203-94.

[25] S. Mahlke, W. Chen, P. Chang, and W. Hwu, "Scalar Program Performance on Multiple-Instruction-Issue Processors with a Limited Number of Registers", Proceedings of the 25th Annual Hawaii International Conference on System Sciences, 1992.

[26] S. W. Melvin, M. C. Shebanow and Y. N. Patt, "Hardware Support for Large Atomic Units in Dynamically Scheduled Machines", Proceedings of Micro-21, San Diego, December 1988, pp. 60-66.

[27] E. Rotenberg, S. Bennett and J. E. Smith, "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching", Proceedings of MICRO- 29,December 1996, Paris, France, pp. 24-34.

[28] Y. Sazeides and J. Smith, "The Predictability of Data Values" Proceedings of MICRO-30

[29] M. Slater, "AMD's K5 designed to outrun Pentium", Microprocessor Report, 8(14):1-11, October 1994.

[30] A. Sodani and G. S. Sohi, "Dynamic Instruction Reuse", Proceedings of the International Symposium on Computer Architecture, 1997, pp. 194-205.

[31] G. Steven, "A Novel Effective Address Calculation Mechanism for RISC Microprocessors", Computer Architecture News, pp. 150-156, September 1988.

[32] M. Smotherman and M. Franklin, "Improving CISC Instruction Decoding Performance Using a Fill Unit", Proceedings of Micro-28, 1995, pp. 219-229.

[33] J. E. Thornton, “Design of a Computer: The Control Data 6600”, Scott, Foresman and Company, Glenview 1970.

[34] K. Wang and M. Franklin, “Highly Accurate Data Value Prediction using Hybrid Predictors”, Proceedings of MICRO-30

[35] S. Weiss and J. E. Smith, “ POWER and PowerPC: Principles, Architecture, Implementation”, Morgan Kaufman Publishers, 1994.

[36] L. Widigen, E. Sowadsky and K. McGrath, “Eliminating Operand Read Latency”, Computer Architecture News, December 1996, pp. 18-22.

[37] J.E. Wilson, S.W. Melvin, M.C. Shebanow, W.M. Hwu, and N.Y. Patt, “On Tuning the Microarchitecture of an HPS Implementation of the VAX,” Proc. Micro-20, Colorado Springs, December 1987, pp. 162-167.

[38] PentiumPro Processor: A Tour of the microarchitecture, Intel web site, http://www.intel.com/procs/p6/p6white/p6white.htm

[39] Private Communication with xxxxxxxx, Intel Corporation. (name suppressed for anonymous review)

[40] Private Communication with xxxxxxxx, Intel Corporation. (name suppressed for anonymous review)