Signature Buffer: Bridging Performance Gap between Registers and Caches

Lu Peng Jih-Kwon Peir
Computer Information Science & Eng.
University of Florida
{lpeng,peir}@cise.ufl.edu

Konrad Lai
Microprocessor Research Lab.
Intel Corporation
konrad.lai@intel.com

Abstract

Data communications between producer instructions and consumer instructions through memory incur extra delays that degrade processor performance. In this paper, we introduce a new storage media with a novel addressing mechanism to avoid address calculations. Instead of a memory address, each load and store is assigned a signature for accessing the new storage. A signature consists of the color of the base register along with its displacement value. A unique color is assigned to a register whenever the register is updated. When two memory instructions have the same signature, they address to the same memory location. This memory signature can be formed early in the processor pipeline. A small Signature Buffer, addressed by the memory signature, can be established to permit stores and loads bypassing normal memory hierarchy for fast data communication. Performance evaluations based on an Alpha 21264-like pipeline using SPEC2000 integer benchmarks show that an IPC (Instruction-Per-Cycle) improvement of 13-18% is possible using a small 8-entry signature buffer.

1. Introduction

Program execution obeys a sequence of instructions that operate on a collection of data. Data communication among instructions usually goes through storages that save intermediate values from a producer instruction for future consumer instructions. Two types of storages, registers and memory, are available in a processor. A small number of architecture registers can supply data for operations seamlessly. Memory, on the other hand, is large and slow, even with caches that enable fast accesses for recently used data. Due to the disparity of speeds, modern processors permit instructions to only directly operate on data from registers, and use loads and stores to move the data between memory and registers. Therefore, data communication among instructions goes through an indirect path: *producer → store → load → consumer.*

In spite of mapping as many program locations to limited registers as possible by the compiler, it is inevitable to use memory for data communication. Due to a load often encountered right before the consumer needs for efficient usage of registers and its inherent delays in generating the address and accessing large storages, loads along with their dependent instructions are likely located on the program's critical path [22]. The performance penalty of load latency will worsen in future processors. As the feature size continues to shrink and clock speed approaches 10 GHz, it is estimated that the access time of a 64KB first-level cache (L_1) will take three to seven cycles according to different clock scaling factors using a 35nm technology [1]. Simulation studies show that each additional cycle in accessing the L_1 data cache degrades the overall IPC by about 3.5% on an Alpha 21264-like pipeline running SPEC2000 integer programs.

To achieve a *zero-cycle* load, i.e. the load and its dependent instructions can be fetched, dispatched and executed at the same time; we introduce a new storage structure that can be accessed in early pipeline stages. The new storage has a very small physical structure to enable fast access time. In addition, it uses a novel addressing mechanism to avoid any address calculation. Each load and store uses a *signature* for accessing the new storage. A signature consists of the *color* of the base register along with the *displacement* value. Each color represents a unique instance of the base register. A new color is given whenever the content of the register is updated. When two memory instructions have the same color, they address memory locations with the same base register and base address. In this case, the displacement value correctly identifies the data item with respect to the base address. A small *Signature Buffer (SB)* addressed by the memory signature can be established to provide data for memory operations *non-speculatively*. Accessing the SB is initiated early in the pipeline after a memory instruction is fetched and decoded since the signature can be formed without accessing the register file.

Memory dependences or *correlations* of store-load or load-load are preserved with signatures. This property exists in many program constructs such as accessing global

and local variables, saving/restoring registers during procedure/function calls, referencing structure records using pointers in linked data structures, or accessing array elements in loop iterations. Performance evaluations based on the SPEC2000 integer benchmarks running on an Alpha 21264-like processor model show that 70% of the loads can benefit from a small 8-entry SB to reduce access latency. This latency reduction translates to about 13-18% IPC (Instruction-Per-Cycle) improvement.

The remaining paper is organized as follows. The motivations and important observations for the proposed method will be described in section 2. This is followed by discussions of design and related issues for establishing the SB in Section 3. In Section 4, performance evaluations of the IPC improvement as well as the SB hit ratios are given. Several design parameters and alternatives for the SB are also evaluated. A few related works on hiding cache latency will be given in Section 5. Finally, Section 6 concludes the paper.

2. Memory Reference Correlations

Store-load and load-load memory dependences can be correlated directly by the signature if they use the same base register with the same displacement value as long as the base registers have an identical content (i.e. the same color). Memory references also exhibit strong spatial locality using the signature as nearby memory references often differ only by a small quantity of the displacement value. In this section, we provide two programming examples from SPEC2000 integer programs to describe qualitatively the existence of such store-load and load-load *signature correlations* and *signature reference locality* in real programs. In Figure 1, an example function *copy_disjunct* from Parser is given. This function is invoked many times to build a new copy of a disjunct list. The second example *bsW* is extracted from Bzip (Figure 2). This function is also invoked multiple times to perform bit-stream I/Os. The store/load signature correlation and reference locality can be observed in several program constructs.

Access Records in Linked Data Structures: As shown in Figure 1, the pointers *d, d1* are used to copy and construct a new node in the linked structure. Different records (also pointers in this case) in each node of the old and the new linked structures are accessed using pointers *d, d1*. In the assembly code, the two pointers are loaded in registers $s0, $s1 and are used as the base registers to access these variations of records with small displacement values. The signature correlation and reference locality among these memory accesses are clearly demonstrated.

Register Save and Restore in Functions: In example I, strong spatial locality exists when restoring register contents in function copy_disjunct. In addition, store-load signature correlations exist with matched base register $sp and displacement value for saving and restoring register contents without intervening function calls. The invocations of *xalloc* and *copy_connectors* may change the color of the $sp, but the original content of the $sp is restored after returning from the function calls to address caller's stack frame.

```
Disjunct * copy_disjunct(Disjunct * d) {
   Disjunct * d1;
   if (d == NULL) return NULL;
   d1 = (Disjunct *) xalloc(sizeof(Disjunct));
   *d1 = *d;
   d1->next = NULL;
   d1->left = copy_connectors(d->left);
   d1->right = copy_connectors(d->right);
   return d1;
}


copy_disjunct:
     addiu $sp,$sp,-32
     sw    $s1,20($sp)
     addu  $s1,$0,$a0
     sw    $ra,24($sp)
     sw    $s0,16($sp)
     beq   $s1,$0,<copy_disjunct>
     addiu $a0,$0,20
     jal   <xalloc>
     addu  $s0,$0,$v0
     lw    $v0,0($s1)
     lw    $v1,4($s1)
     lw    $a0,8($s1)
     lw    $a1,12($s1)
     sw    $v0,0($s0)
     sw    $v1,4($s0)
     sw    $a0,8($s0)
     sw    $a1,12($s0)
     lw    $v0,16($s1)
     sw    $v0,16($s0)
     sw    $0, 0($s0)
     lw    $a0,12($s1)
     jal   <copy_connectors>
     ......
     lw    $ra,24($sp)
     lw    $s1,20($sp)
     lw    $s0,16($sp)
     addiu $sp,$sp,32
     jr    $ra
```

Register save/ restore

Access linked records

Figure 1. Example I - Source and Assembly Codes of Function *copy_disjunct* from Parser

Access Array Variables: Similar store/load correlations are also observed in accessing array data structures in other program code. For example, intensive array accesses are observed in several functions, such as *_word_fwd_aligned()* in *wordcopy.c* in Gcc of SPEC2000. Nearby references to the same or different elements of the same array with the same base address provide signature correlations of stores and loads.

Access Global Variables: As shown in Figure 2, three global variables, *bsBuff*, *bsLive* and *bytesOut* are accessed when the function *bsW* is invoked. Due to the limited registers, these variables are loaded/stored multiple times based on the same global pointer $gp with a constant base

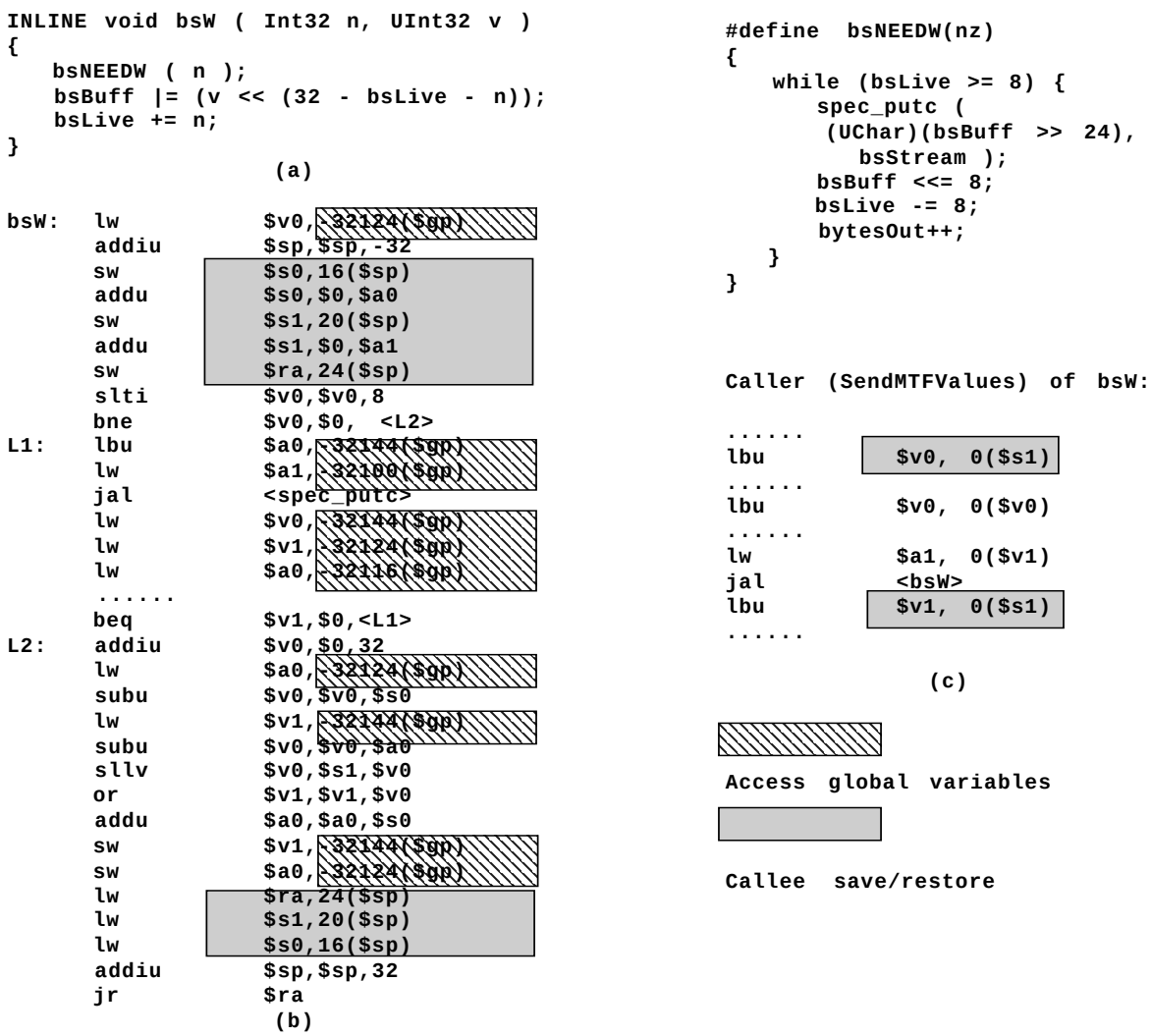

Figure 2. Example II - Source and Assembly Codes of Function bsW from Bzip. (a) Source Code; (b) Assembly Code; (c) Caller of bsW

address. The access of global variables exhibits both signature correlations and spatial locality.

Access Local Variables: In the *bsW*, the callee-saved registers $s0 and $s1 are freed up for local usages to avoid saving parameters of *n* and *v* from registers $a0 and $a1 to the local stack frame and retrieving them later for computations. However, in functions that involve more complex computations and/or more temporary local variables, it is inevitable to increase the local stack accesses using the stack pointer $sp and/or the frame pointer $s8 as base registers. Such accesses to the small stack frame usually display signature correlations and spatial locality.

General Save/Restore Base Registers: There is evidence that even if the base register has been updated between two memory accesses, the content of the base register may stay the same. A base register may be freed up for other usages and the original base address is restored before the next memory reference. In Figure 2, we also include a partial assembly code from a caller *SendMTFValues* of the *bsW*. In this caller, $s1 is used as a base register before calling the *bsW*. After returning from the *bsW*, $s1 is restored and continues to be used as a base register.

Passing Pointers among Multiple Functions: We also observe in other programs, e.g. Mcf, that register class $a is sometimes used to pass pointers among several levels of function calls. The pointer in $a is used as the base register in several functions without any save/restore the content of $a. As a result, the same color for $a is maintained across multiple functions to keep the signature correlation alive.

3. Establishing A Signature Buffer

In Figure 3, a load is used to illustrate the basic design of the *Signature buffer (SB)*. Besides a small, fully-associative SB, correct signatures for memory instructions can be established using a Color Register (CR) and a Current Color Table (CCT). The CR keeps the next available color and the CCT records the current color of each register. Initially, each CCT entry is set to the corresponding ID (from 0 to 31) of each register, while the content of CR is set to 32 (assuming 32 base registers). When a load is decoded in-order, the current color of the base register (R1) is fetched from the CCT. The color is concatenated with the displacement value from the encoding bits of the load to form the signature. Meanwhile, the new color of the destination register (R2) is updated from the CR and the CR is incremented afterwards. This new color assignment takes place for all instructions that involve a register update. To guarantee correct data access, all assigned colors along with the SB are flushed when the color has reached its maximum value and is wrapped around to the initial value. Essentially, a base register is renamed to a new color upon an update to the register. When two memory moves have the same color, they always have the same base address. Note that this coloring technique conservatively assigns a register value without accessing the register file to avoid delays and any stall due to base register updates.

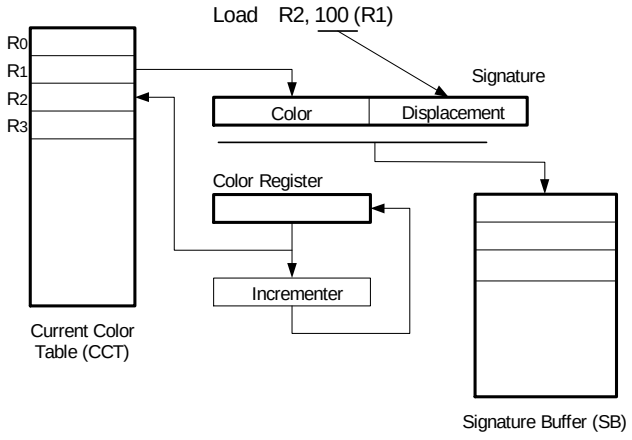

Figure 3. Memory Signature and Signature Buffer

The signature buffer is always accessed in-order after the signature is formed. When the signature of a load matches the signature of a line in the SB, the data can be obtained non-speculatively from the SB. This is similar to a virtually addressed cache such that a match of the virtual address guarantees the correct data found in the cache [6]. The SB can be thought as a virtual cache using the signature as the virtual address. Since the SB is accessed right after the load is decoded, it is conceivable that load dependents can also be fetched and issued without any delay. Several design issues of the SB will be discussed in subsequent sub-sections.

3.1. Data Alignment

The low-order bits in the signature, i.e. the offset bits within a signature line, may not be the same as the offset bits in the real memory address. In order to exploit the spatial reference locality, the cache line fetched from the first-level cache (L_1) needs to be rearranged in the SB to align with the memory signature. The basic alignment algorithm works as follows. When a memory request misses the SB, the target cache line is fetched. The target byte/word is placed in the SB according to offset bits of the signature. For example, assume there are eight units in a cache line as shown in Figure 4. The signature offset of the target unit is 001, but the offset of the real address is 100. In this case, the target data 100 is loaded into unit 001 in the SB, and remaining units are loaded according to their relative position to the target unit. Each SB line may contain a part of two consecutive L_1 lines. We refer these two parts as the *High* and the *Low* partitions. In the example in Figure 4, the partial SB line fill is located in the *Low* partition of the SB line.

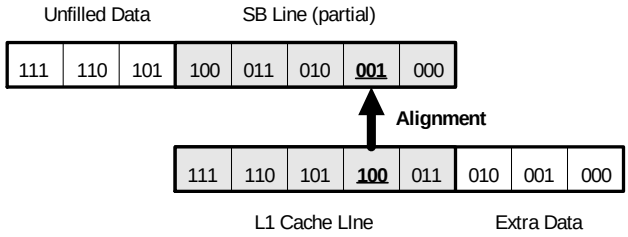

Figure 4. Data Alignment in Signature Buffer

Due to the alignment, there are two options to fill an SB line. The straightforward option is to fill only a partial SB line and drop the extra data from the target L_1 line. Other options include fetching the second L_1 cache line to fill the entire SB line, and/or to place the extra-unaligned L_1 data into the 2nd SB line to further benefit spatial locality.

3.2. Handling Signature Synonym

One essential issue in designing the SB is to handle the synonym of memory signatures when two distinct signatures are mapped to the same memory address. This synonym problem has some similarity with that in virtual caches. In addition to the signature tag, the corresponding L_1 address tag is also saved in the SB directory. After the memory address is generated upon an SB miss, a match through L_1 address tags in the SB can identify any signature synonym. A synonym hit occurs if the line is

located in the SB under a signature with a different color. In this case, the line is invalidated from the old and moved to the new locations. Given that the proposed SB is fully-associative with small number of lines, the SB synonym can simply be managed by snooping the SB on every replacement. Virtual caches with set-associative design require extra hardware to resolve synonyms.

However, signature synonym is complicated by the fact that each signature line may be aligned to have a portion of two consecutive L_1 cache lines. Care must be taken to distinguish a synonym hit and to invalidate the line from two possible SB locations. The detailed procedure is described as we walk through the following example.

In Figure 5, the contents of the SB along with the data alignment between the SB and the L_1 cache lines are illustrated. The SB directory records the signature tag, the L_1 address tag, the validity bits, and the boundary of High and Low partitions. An SB hit is identified when the tag matches and the target High/Low partition is valid. The boundary of the partition determines the High/Low partition. Recall that due to the data alignment, the High/Low partitions indicate two data regions from two L_1 cache lines. These two lines must be consecutive because each L_1 line cannot exist in the SB with two different colors (i.e. a synonym case). Therefore, only a single L_1 address tag is recorded in the SB directory.

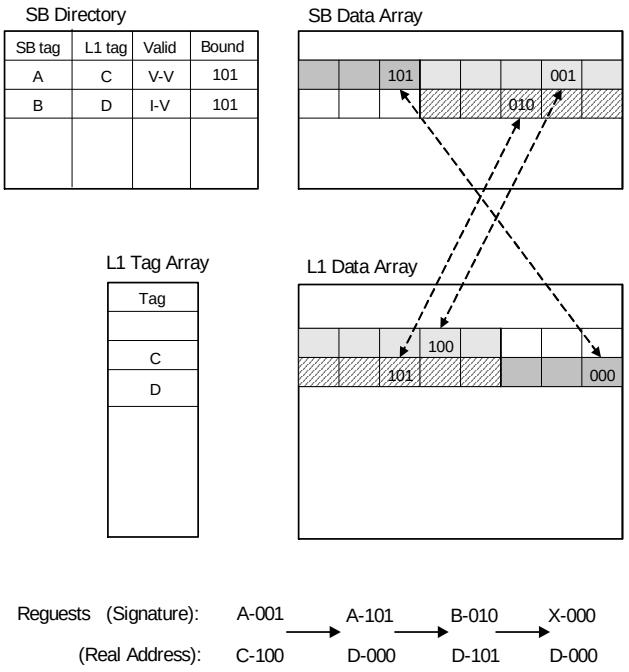

Figure 5. Tag Arrays and Data Alignment between SB and L_1 Cache for 4 Consecutive Memory Requests; A, B, X Are Signature Lines; C, D Are L_1 Cache Lines; Three Binary Bits Are Target Unit; "Bound" Denotes Boundary of High / Low Partitions

Assume that three consecutive memory requests with signatures: A-001, A-101, and B-010 are issued, where the binary bits represents the target unit in each signature line. All three requests have an identical color, and A, B are two consecutive signature lines. These requests are mapped to the corresponding L_1 lines: C-100, D-000, and D-101 in the L_1 cache, where C and D are adjacent lines. Based on the data alignment, the most significant 5 units of line C are loaded to the SB for the first request. A simple 2's compliment subtraction of the L_1 unit ID from the SB unit ID can determine the High/Low partition and the correct boundary. For the first request, *(0,001 - 0,100 = 0,001 + 1,100 = 1,101)*; the result sign bit '1' indicates the partition is 'Low', and the magnitude '101' is the boundary. (Detailed arithmetic reasoning is omitted.) The L_1 address tag, the corresponding valid bit, and the boundary 101 are recorded in the SB tag array.

The second request is also a miss even though tag A matches because the target unit 101 is in the invalid partition. Only the less significant 3 units of L_1 line D are loaded to the most significant 3 units of signature line A. Both valid bits of the High/Low partitions of A are now set. The third request addresses a different signature line B, and causes another miss. Again, according to the alignment, the most significant 5 units of line D are loaded to the lower 5 units of line B in the SB. Next, the fourth request to a signature line X is issued and misses the SB. This request is mapped to D in the L_1 cache with a different color than that of A and B. Now, a synonym hit is detected and D must be invalidated from the SB. Two invalidations are needed, one on the High partition of A and the other on the Low partition of B.

3.3. Memory Dependence Disambiguation and Data Forwarding

The SB and the L_1 cache are updated when stores are committed. To shorten memory dependence latency, the stored data can be forwarded to dependent loads before the store is committed. This store-load forwarding through the Load Store Queue (LSQ) has been applied to bypass the L_1 data cache. The store-load dependence can be resolved based on the address comparison. This dependence resolution is more critical with the SB for two reasons. First, upon an SB hit, a load may have to wait for any uncommitted store ahead of the load in the LSQ. The uncertainty of memory dependences may hinder the load and degrades the advantage of fetching data from the SB. Second, early resolution of store-load forwarding allows bypassing the SB. Using the signature instead of the address, certain memory dependences can be resolved in parallel with the SB access. Several steps using an expanded LSQ to alleviate this memory dependence delay are summarized below.

• **Early store-load forwarding:** When a load has an

identical signature with an early store in the LSQ, and there is no intervening store in between, the load bypasses the SB and gets the data from the early store. An intervening store has a different color and is a potential synonym to the load.

• **Early SB access with memory disambiguation:** The SB is accessed after a load is fetched and decoded. The data obtained from the SB can trigger dependent instructions without delays if there is no intervening store ahead of the load in the LSQ.

• **Delayed SB access:** The SB is successfully accessed after memory dependence resolutions because of an existence of intervening stores

• **Forwarding from early SB misses:** When consecutive SB misses are to the same SB line, the later miss gets the forwarded data from the early miss. This forwarding can happen before the load address is computed. (This case is considered as a partial SB hit.)

• **Handling multiple SB misses to the same L_1 line:** When multiple SB misses with different colors to the same L_1 line are detected through the LSQ search, later misses are blocked from accessing the L_1 cache and the requested data will be forwarded when the first miss data is available from the L_1.

• **Normal store-load forwarding:** When a load or a store address is computed, a search through the LSQ to identify the store-load forwarding is performed as the normal store-load forwarding.

For better performance, aggressive early store-load forwarding and speculative use of the SB data are possible by predicting the existence of memory dependences with the cost of recovery upon mis-predictions.

3.4. Integrated Pipeline Microarchitecture

The SimpleScalar pipeline is modified and expanded to accommodate performance studies of the SB [4]. Figure 6 shows the baseline microarchitecture, which is more in-line with the Alpha 21264 [13]. The dispatch stage in the SimpleScalar pipeline is partitioned into a *Decode/Dependence* and a *Rename* stages. In the *Decode/Dependence* stage multiple instructions are decoded, and dependences among instructions in the decode/issue packet are detected. The decoded instructions (micro-ops) are renamed to reorder buffer (RUU) entries at the *Rename* stage. The original issue/execute stage becomes a *Schedule*, a *Register* read, and an *Execute* stages to reflect the delays in scheduling, operands fetching, and execution.

The two fast paths in the figure illustrate advantages with an integrated SB. A load can jump to the *Writeback* stage after the *Rename* stage on an SB hit or an early store-load forwarding as indicated by the Bypass I. Here we assume the SB access is completed in the *Decode/Dependence* and the *Rename* stages. The Bypass II denotes a normal store-load forwarding. An SB miss may get the data from an early SB miss for the same L_1 cache line, or a successful SB access may be delayed until intervening store dependence is cleared. Such bypasses, not shown in the figure, could be from the *Schedule* stage all the way to the *Memory* stage. Note that the Bypass I may start as late as the *Register* stage (i.e. with a 4-cycle SB) and still achieve a zero-cycle load since dependent instructions do not start execution until after the *Register* stage. However, an aggressive schedule of loads to the L_1 cache may be needed if the SB access cannot be resolved on or before the *Schedule* stage.

The proposed SB can be integrated in the baseline microarchitecture as shown in Figure 7. In the *Decode/Dependence* and the *Rename* stages of the baseline design, a load is decoded, the signature is formed, and the SB is accessed. Meanwhile, a search through the LSQ for memory dependences with early stores is carried out. If store-load forwarding is found based on matching load signature with an early store without any intervening store in between, the SB is bypassed and the stored data from the LSQ satisfies the load. Similarly, if the requested data is found in the SB and there is no intervening store, the load will jump to the *Writeback* stage to trigger dependent instructions.

A load is moved to the *Schedule* stage when the load misses the SB without store-load forwarding, or an uncertainty of memory dependence exists. The load moves on with fetching the base register (the *Register* stage), and generating the memory address (the *Execute* stage). After resolving memory dependences, the data may be obtained from the LSQ through forwarding. In cases of memory dependence free, the load may catch the data found during an early SB access, or access the L_1 cache normally. A load

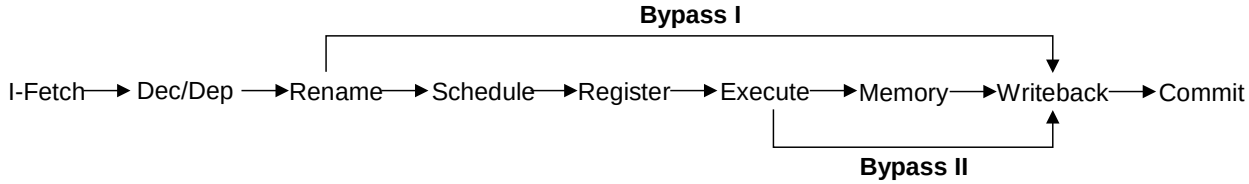

Figure 6. The Pipeline Stages and Bypassing

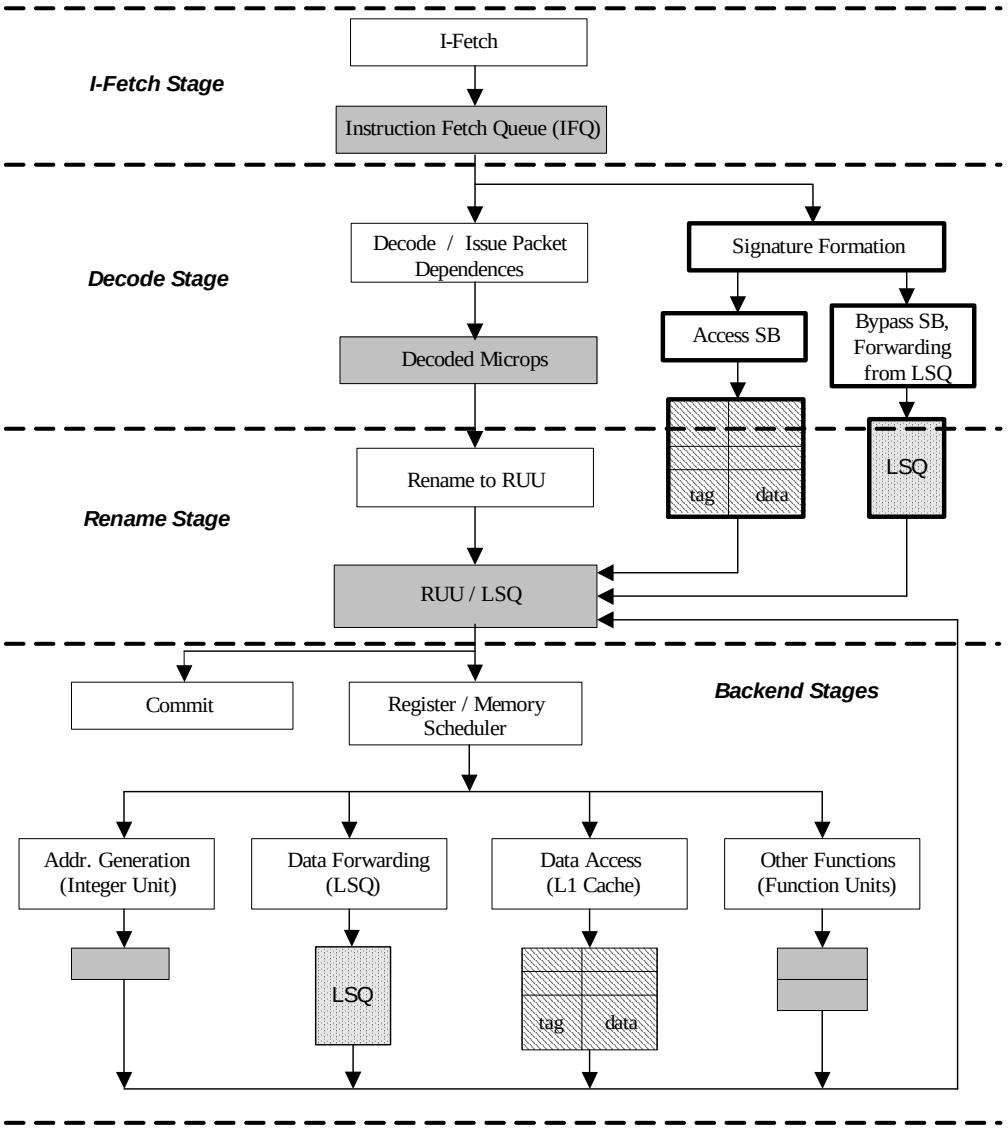

Figure 7. Pipeline Microarchitecture with an Integrated SB

may also wait in the LSQ for an early miss to the same L_1 cache line with the same or different signature.

At the *Writeback* stage, the SB is filled with the target L_1 cache line and the directory is updated in response to the SB load miss. A backward invalidation to the SB is performed upon a synonym hit, or a replacement of a L_1 cache line. Finally, at the *Commit* stage, stores will lookup both the SB and the L_1 cache. The SB and the L_1 cache are updated accordingly.

4. Performance Evaluation

Performance evaluations of the proposed SB are carried out on a modified out-of-order SimpleScalar model [4] as described in Section 3.4. This Alpha 21264-like processor is capable of issuing 8 instructions per cycle. Table 1 summarizes simulation parameters. Note that we assume the SB or a small L_0 has 4 ports with a single-port L_1. For the baseline model, a 4-port L_1 is simulated. Nine integer programs, Bzip, Gap, Gcc, Gzip, Mcf, Parser, Twolf, Vpr, and Vortex from SPEC2000 are chosen. A version 2.7.2.3 *ssbig-na-sstrix-gcc* compiler with option: *(funroll-loops -O2)* generates the binary codes. With the optimization level "*-O2*", register allocations based on graph coloring techniques are applied. For each workload, we skip certain instructions based on studies done in [20], and then collect statistics from the next 200 million instructions.

The out-of-order SimpleScalar pipeline is a functional-driven timing simulation model [4]. For studying the SB, we add value checking in the timing model. The actual value is loaded into the SB from the simulated memory. The value of loads and stores from functional simulations is attached throughout pipeline stages. For loads, the functional value is used to verify the value obtained from the SB. For stores, the functional value is used to update the SB after the store is committed.

TABLE 1
SIMULATION PARAMETERS

Fetch/Decode/Issue Width	8
Branch Predictor	Bimodel, 8K-entry, 8-way BTB
RUU/LSQ Size	64 / 32
SB/L_0 Size	4-16 64-byte line
L_1 Instruction/Data	64KB, 4-way, 64-byte Line
L_2 Cache	2MB, 4-way, 64-byte Line
Latency: SB/L_0/L_1/L_2/Mem	2 / 2 / 5-10 / 10 / 200
Memory Port	4
Integer ALU: Add/Multiply	4 / 2

Several design parameters are considered for the SB. First, we simulate 16-bit and 24-bit colors. Both results are very similar. Secondly, preliminary studies show that the option of fetching the second L_1 cache line to fill the entire SB line, and/or to place the extra-unaligned L_1 data in to the 2nd SB line, slightly improves the SB hit ratios. For design simplicity, we only consider to fill a partial SB line and drop the extra L_1 data. Thirdly, the L_1 topology is fixed at 64KB, 4-way set-associative, with variable delays from 5 to 10 cycles. We simulate small SBs with 4 to 16 lines. The SB access takes 2 cycles as described in Section 3.4. We also verify that a 3- or 4-cycle SB has virtually the same performance improvement. Lastly, for comparisons, we simulate a L_0 cache with equivalent size and access time to that of the SB. The SB, L_0, L_1 and L_2 all have 64-byte line size.

4.1. IPC Improvement

In Figure 8, the IPCs of the nine programs and their arithmetic means are plotted. Three mechanisms are compared: the original baseline model (*Baseline*), the pipeline model with an integrated SB but without any speculation on memory dependences (*SB-nospec*), and the pipeline model with an integrated SB and a perfect memory dependence predictor (*SB-perfect*). A perfect predictor allows loads to bypass any stores in the LSQ as long as there is no dependence between them. In this set of simulations, an 8-entry SB with 2-cycle access is reported and the L_1 cache latency is 5 cycles.

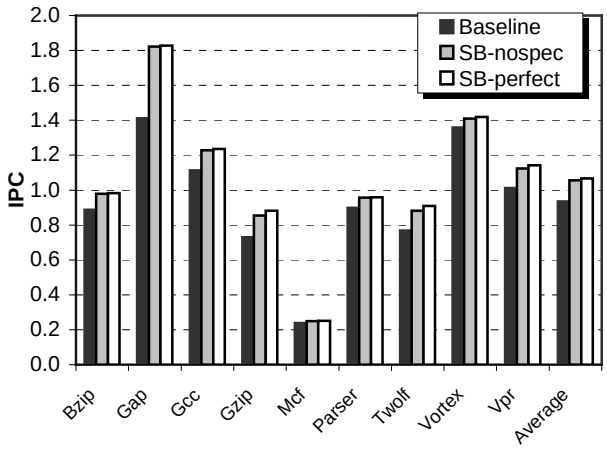

Figure 8. Comparison of IPCs: Baseline without SB; SB without Memory Dependence Speculation; and SB with A Perfect Memory Dependence Predictor

On the average, the *SB-nospec* and the *SB-perfect* improve about 13.0% and 14.2% of IPCs over the *Baseline*. Among the nine programs, Gap and Gzip have the most improvement about 30% and 20% for the SB schemes, while Mcf, Parser and Vortex show poor improvement about 4-7% over the *Baseline*. The impact of a perfect predictor is very limited because the intervening stores can be resolved quickly. Mcf has low IPCs due to its heavy L_2 misses for pointer-chasing references [8].

To understand the impact on IPCs, we show the distribution of loads based on when and where the load data is ready in Figure 9. There are six categories of loads that match the classes in Section 3.3: (1) the data forwarded from a prior store in the LSQ bypassing the SB; (2) the data obtained from the SB without intervening stores; (3) the data forwarded from an early SB miss to the same signature line or obtained from the SB after memory dependence resolutions of intervening stores; (4) the data forwarded from a prior SB miss to the same L_1 line with different colors; (5) the data from a normal store-load forwarding; and (6) the data from a normal L_1 cache access. The first four categories are the beneficiaries of the SB. Among the four, the first two can obtain the load data in the *Rename* stage to achieve a zero-cycle load. The category (3) loads may get the forwarded data before their addresses are computed, while category (4) loads can only receive the forwarded data after the load address becomes available.

We can make several observations from the figure. First, using the SB, the normal store-load forwarding and the L_1 access are reduced to about 30%. In other words, 70% of the loads benefit from the proposed scheme to obtain data sooner than accessing it from the L_1 data cache. The perfect predictor improves the category (2) loads, i.e. the data from the SB in the *Rename* stage, from 7% to 23% due to bypassing the intervening stores. Gzip is

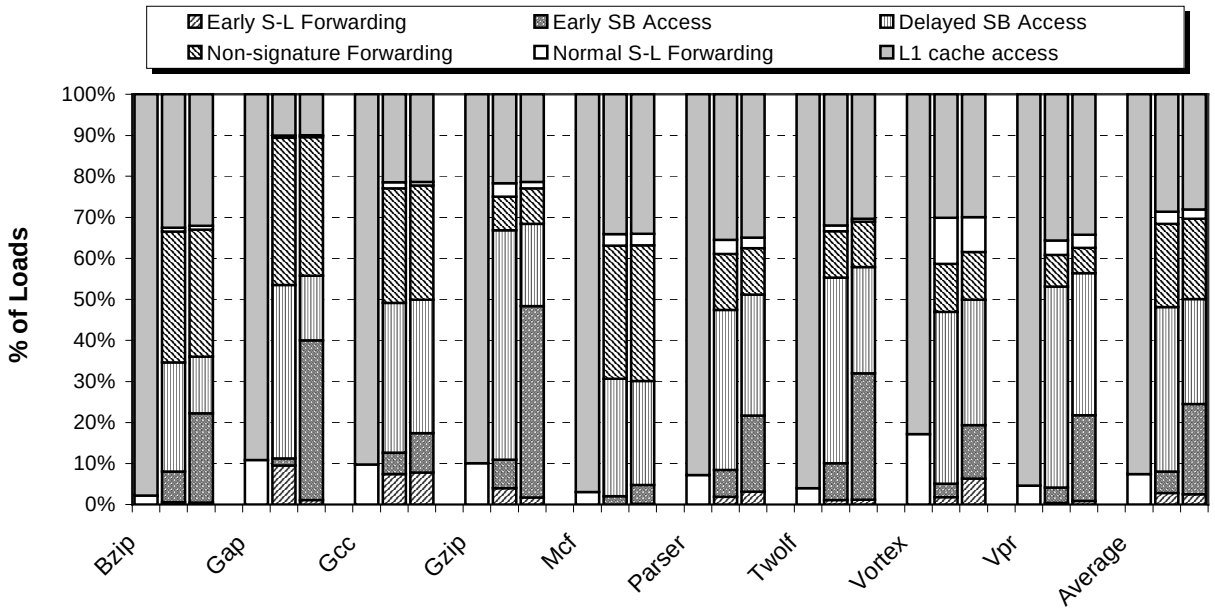

Figure 9. Distribution of Loads (Six Categories from Bottom to Top in Each Bar): (1) Early Store-Load Forwarding; (2) Early SB Data Access; (3) Forwarding from Early SB Miss or Delayed SB after Memory Dependence; (4) Forwarding from Prior SB Miss to Same L_1 Line; (5) Normal Store-Load Forwarding; (6) Normal L_1 Cache Access

most beneficial using a perfect predictor with close to 50% of category (2) loads. Detailed analysis indicates that about 52% of the loads in Gzip are accessing global variables with rarely any true synonym cases. Gap has 40% category (2) loads with 35% global accesses.

Second, from the *SB-nospec*, 40% of the loads obtain data either from an early SB miss to the same SB line (and the same High/Low portion), or from the SB after resolving intervening memory dependences. Memory references exhibit strong spatial locality such that misses to the same SB line are often close to each other. For example, in Figure 1 and 2, many adjacent loads' signatures differ only by a small quantity of the displacement value. As a result, the later loads can receive data directly from an early L_1 access. A typical case is found in function *_word_fwd_aligned()* in *wordcopy.c* of Gcc, where this type of consecutive loads represents 54% of the total executed loads. Under the *SB-perfect*, however, these category (3) loads are reduced to 25% because of the reduction of delayed SB accesses. Again, Gzip and Gap have high percentage of global accesses with rarely any signature synonym.

Third, an average 20% of the loads belong to category (4) that indicates many short bursts of SB misses to the same L_1 cache line with different colors. This situation is encountered because of signature synonyms among nearby loads. For example, in accessing array elements, the base register is incremented by a small stride for each access. Although the color changes, several array accesses may still address the same L_1 line and they all potentially miss the SB. This access type represents pre-dominated loads we found in function *ProdInt()* in *integer.c* of Gap. A similar case may also be encountered due to spilled codes with limited registers. Base registers are saved and restored to the original base address before the next usage. As a new color is assigned on each register update, nearby loads may miss the SB, but address the same L_1 line.

Fourth, in general, the IPC improvements of Figure 8 are consistent with the categories of loads in Figure 9. For example, Gap, Gcc, Gzip, and Twolf have more loads that benefit from the SB, while Mcf, Parser, and Vortex have smaller amount of beneficial loads. Note that the overall IPC impact is beyond just the pure reduction of load latencies. Improving load latency may not help the IPC if the load is not located in the critical execution path [22].

4.2. SB Hit Ratio

We collect the SB hits/misses during cycle-based simulations with 8-entry, 2-cycle SBs and 5-cycle L_1 access latency. The SB hit/miss is determined at the *Decode/Dependence* cycle regardless an existence of any memory dependence. A load is considered as an SB hit if it hits the SB, or it misses the SB, but an early miss to the same SB line (and the same High/Low partition) is already in the LSQ. In Figure 10, we break down SB hits and misses with respect to the base register IDs. We separate base registers into 5 groups: $v, $a, $s+$t, $gp, and

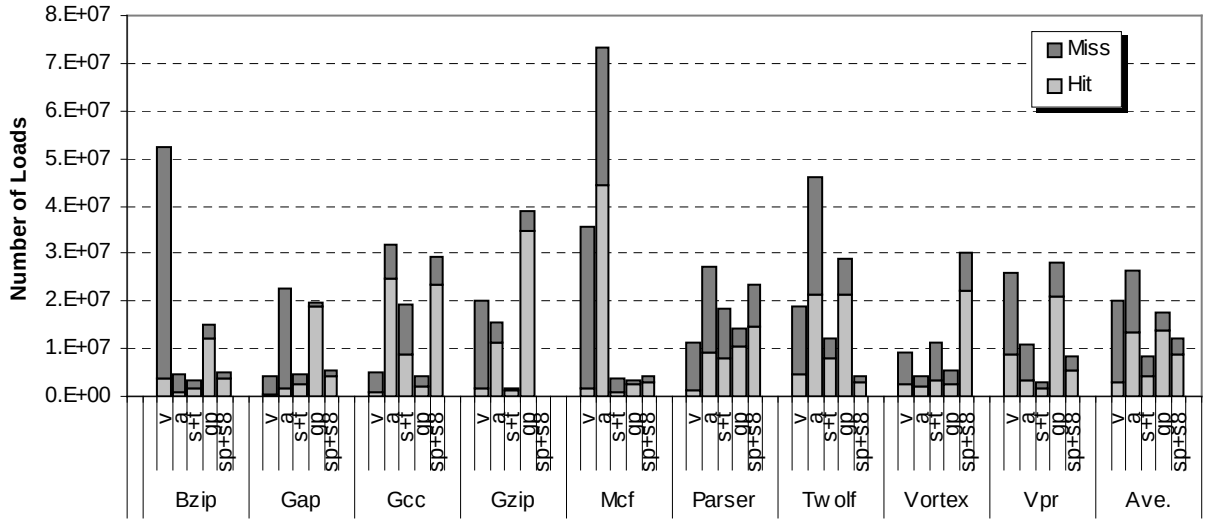

Figure 10. SB Hit Ratios Broken Down by Base Registers

$sp+$s8. The overall SB hit ratio is about 51%. Mcf is dominated by pointer-chasing references using $a as the base. A portion of these references access records in linked structures or pass pointers among multiple functions. The SB scheme handles these type references reasonably well with about 63% hit ratio for $a.

The distribution and accuracy of individual groups are shown in Table 2. As expected, it is quite accurate to access global variables and local stack frames. For other loads, the compiler first picks $v and $a as temporary registers to hold base addresses for memory accesses. The base address is often updated right before the load that may results in a miss from the SB.

TABLE 2
DISTRIBUTION AND SB HIT RATIOS ACCORDING TO BASE REGISTER GROUPS

Base Regester	$v	$a	$s+$t	$gp	$sp+$s8
Distribuation (%)	23.9	31.5	10.0	20.3	14.3
SB Hit Ratio (%)	13.9	51.5	46.7	79.5	71.9

The average SB hit ratio of 51% matches well against the load distributions among 6 categories in Figure 9. It has shown that about 50% of the loads benefit from the SB hit (category (1) through (3)).

4.3. Sensitivity Studies and L_0 Comparison

The impact of the L_1 cache latency is shown in Figure 11. The IPCs for the three caching mechanisms are plotted with respect to the L_1 cache latency of 5 to 10 cycles. From the *Baseline*, each cycle reduction on L_1 cache access latency improves the IPC by 3.5% (slightly lower than the original SimpleScalar pipeline). The performance benefit of the SB goes up with the L_1 cache latency as we expected. The IPC improvements for the *SB-nospec* are ranging between 13-17% for 5 to 10 L_1 cache latency cycles. For the *SB-perfect*, the improvements are about 14-18% over the simulated baseline model.

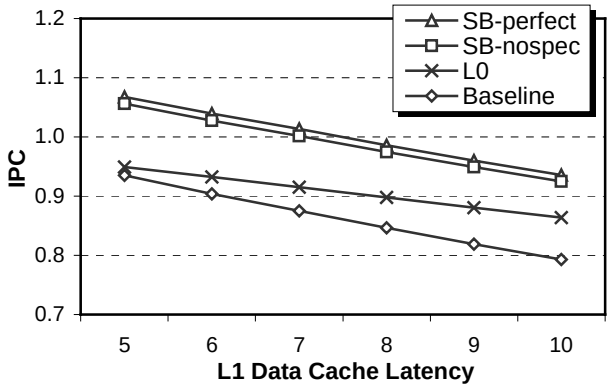

Figure 11. Average IPC for Various L_1 Cache Latencies

To prove the key advantage of early SB accesses, we also simulated a L_0 cache with equivalent size (8 lines) and access time to that of the SB. As shown in Figure 11, the SB out-performs the L_0 by 7-11% without a perfect memory dependence predictor. L_0 can only be accessed after the address is computed. Therefore, with a 2-cycle access L_0, a 3-cycle latency is imposed to the dependent instruction. Furthermore, unlike the SB, L_0 miss causes extra delay to access the L_1 cache.

We simulate small SB and L_0 sizes with 4, 8, 12, and 16

lines. We also simulate fast and slow SB, L_0 and L_1 caches. For fast caches, we assume 2-cycle SB/L_0 and a 5-cycle L_1, while for slow caches, we assume 3-cycle SB/L_0 and a 8-cycle L_1. The results in Figure 12 suggest that the SB size plays a very minor role in improving the IPC. Basically, the working set between base register updates is small with strong spatial locality. Although larger SB can keep the data longer, frequent updates of base registers wipe out the correlated data. L_0 caches, on the other hand, show more IPC improvement with bigger size due to higher L_0 hit ratios. However, the SB still shows superior performance with 1KB size. L_0 demonstrates negative IPC improvement with 4 cache lines due to the high miss ratio and additional L_1 access delays.

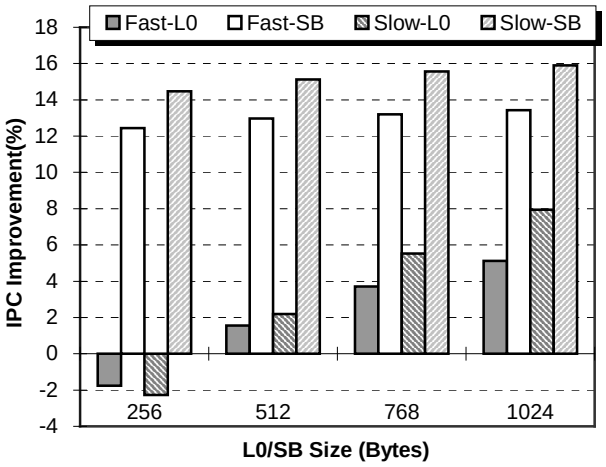

Figure 12. IPC Improvement for Various SB/L_0 Size
Fast: SB/L_0: 2 cycles, L_1: 5 cycles
Slow: SB/L_0: 3 cycles, L_1: 8 cycles

As expected, the SB is more beneficial with slower access speeds since both 2- and 3-cycle SB achieve zero-cycle loads. For L_0 caches, it also shows more advantage with slow speeds because of the longer L_1 latency.

5. Related Work

There have been several attempts to minimize the cache latency penalty by providing a zero-cycle load, i.e. the load and its dependent instructions can be fetched and issued simultaneously. An aggressive approach is to predict and speculate the load value [14,15,24,21,23,11,5], or the load address [9,10,7,2] in early stages of the processor pipeline to issue load dependents without any delay. Both load value and load address predictions generally suffer low accuracy. For address predictions, a lengthy cache access is still required which may delay load dependents even if the predicted load address is correct.

Memory renaming techniques establish dynamic dependence correlations between stores and loads [23]. A separate storage element called a value file is used to save the correlated data. When a memory load instruction is fetched, an indirect access to the value file based on the PC of the load can retrieve the data without going through the lengthy cache access. A similar idea has been use to exploit store-load [17] and load-load [18] correlations. The PC of the load can indirectly access a small synonym file, which keeps the correlated data. Another method of capturing store/load correlations has been proposed recently [16]. Instead of building dynamic dependence correlations, a symbolic cache is used to capture syntax correlations based on the encoding bits from stores and loads.

All above methods are speculative in nature. The speculative load value must be verified against the value obtained from a normal cache access with penalties for mis-predictions. Recently, another technique for early load address resolution was proposed [3]. Certain types of memory loads, such as stack accesses, global constants, or stride-based memory accesses, have regular increment/decrement address patterns. By tracking base registers for this type of loads, register updates can be performed at the decode stage. As a result, address generations and cache accesses for dependent loads can start earlier. Although non-speculative, this approach still needs lengthy cache accesses.

A small level-0 data cache (L_0) [25], a different form of small buffer [12], or a L_0 only for critical loads [19] helps to reduce the load latency. However, since access to the L_0 requires going through normal pipeline stages to decode and generate addresses, it usually cannot achieve 0-cycle loads.

The proposed signature buffer has several advantages over existing cache latency hiding methods. First, different from the symbolic cache, the SB is non-speculative. The data obtained from the SB without intervening stores is always correct. Second, all loads can access the data from the SB without any restriction on the type of the loads or certain special base registers. Third, unlike the address prediction or the register tracking scheme, loads through the SB can bypass the address generation and cache access completely. Fourth, unlike the memory renaming technique, where the store/load correlation is established dynamically by the hardware, the store/load correlation is established from the instruction encoding bits to simplify hardware requirement. Finally, the SB uses line-based granularity to capture the spatial locality among memory references.

6. Conclusion

A non-conventional signature buffer, which is addressed by the signature of store/load instructions, is introduced. A memory signature using the color of the base register and the displacement can be quickly formed to enable data accesses through the SB in early processor pipeline stages. A color of a base register is used to differentiate contents of the same base register. A new color is assigned whenever the respective register is

updated. Therefore, when two memory requests have the same memory signature, they address to the same memory location. The load penalty through regular caches can thus be eliminated when the requested data is obtained from the SB. Performance evaluation of SPEC integer programs has demonstrated that the proposed method can successfully improve the latency for about 70% of the loads. On an Alpha 21264-like processor model, these early loads using the SB can translate to about 13-18% of IPC improvement.

7. Acknowledgement

This work is supported in part by an NSF grant EIA-0073473 and by research donations from Microprocessor Research Lab and China Research Center of Intel Corp. Anonymous referees provide helpful comments.

8. References

[1] V. Agarwal, M. S. Hrishikesh, S.W. Keckler, and D.Burger, "Clock rate versus IPC: the end of the road for conventional microarchitectures," Proc. of 27th Int'l Symp. on Computer Architecture, Vancouver, Canada, June 2000, pp. 248-259.

[2] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz, and U. Weiser, "Correlated Load-Address Predictors," Proc. Of 26th Int'l Symp. on Computer Architecture, Atlanta, GA, May 1999, pp. 54-63.

[3] M. Bekerman, A. Yoaz, F. Gabbay, S. Jourdan, M. Kalaev, and R. Ronen, "Early Load Address Resolution Via Register Tracking," Proc. of 27th Int'l Symp. on Computer Architecture, Vancouver, Canada, June 2000, pp. 306-315.

[4] D. Burger and T. Austin, "The SimpleScalar Tool Set, Version 2.0," Technical Report #1342, CS Department, University of Wisconsin-Madison, June 1997.

[5] B. Calder, G. Reinman, and D. Tullsen, "Selective Value Prediction," Proc. of 26th Int'l Symp. on Computer rchitecture, Atlanta, GA, May 1999, pp. 64-75.

[6] M. Cekleov and M. Dubois, "Virtual-address caches. Part 1: problems and solutions in uniprocessors," IEEE Micro, Vol. 17, Issue 5, Sept/Oct 1997, pp.64-71

[7] C. Chen and A. Wu, "Microarchitecture Support for Improving the Performance of Load Target Prediction," Proc. of 30th Int'l Symp. on Microarchitecture, Triangle Park, NC, Dec. 1997, pp. 228-234.

[8] J. Collins, S. Sair, B. Calder and D. M. Tullsen, "Pointer Cache Assisted Prefeching," Proc. of 35th Int'l Symp. on Microarchitecture, Istanbul, Turkey, Nov. 2002, pp. 62-73.

[9] R. Eickemeyer and S. Vassiliadis, "A Load-Instruction Unit For Pipelined Processors," IBM Journal of Research and Development, Vol. 37(4), pp. 547-564, July 1993.

[10] J. Gonzalez, and A. Gonzalez, "Speculative Execution via Address Prediction and Data Prefetching," Proc. of 1997 Int'l Conf. on Supercomputing, Vienna, Austria, Aug. 1997, pp. 196-203.

[11] J. Gonzalez, and A. Gonzalez, "The Potential of Data Value Speculation to Boost ILP," Proc. of 1998 Int'l Conf. on Supercomputing, Melbourne, Australia, June, 1998, pp. 21-28.

[12] L. John, Y. Teh, F. Matus, and C. Chase, "Code Coalescing Unit: A Mechanism to Facilitate Load Store Data Communication," Proc. ICCD'98, Oct. 1998, pp. 550-557.

[13] R.E. Kessler, "The Alpha 21264 microprocessor," IEEE Micro, Vol 19, Issue 2, Mar/Apr 1999, pp. 24-36.

[14] M. Lipasti, C. Wilkerson and J. Shen, "Value Locality and Load Value Predictionls," Proc. of the 7th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems, Boston, MA, Oct. 1996, pp. 138-147.

[15] M. Lipasti and J. Shen, "Exceeding the Limit via Value Prediction," Proc. of 29th Int'l Symp. on Microarchitecture, Paris, France, Dec. 1996, pp. 226-237.

[16] Q. Ma, J-K. Peir, L. Peng, and K. Lai, "Symbolic Cache: Fast Memory Access Based on Program Syntax Correlation of Loads and Stores," Proc. of 2001 Int'l Conf. on Computer Design, Austin, TX, Sep. 2001, pp. 54-61.

[17] A. Moshovos and G. Sohi, "ioStreamlining Inter-Operation Memory Communication via Data Dependence prediction," Proc. of 30th Int'l Symp. On Microarchitecture, Triangle Park, NC, Dec. 1997, pp. 235-245.

[18] A. Moshovos and G. Sohi, "Read-After-Read Memory Dependence Prediction," Proc. of 32nd Int'l Symp. on Microarchitecture, Haifa, Israel, Nov. 1999, pp. 177-185.

[19] R. Rakvic, B. Black, D. Limaye and J. Shen, "Non-vital Loads", Proc. of 8th Int'l Symp. on High-Perfomance Computer Architecture, Boston, MA, Feb. 2002, pp. 145-154.

[20] S. Sair, M. Charney, "Memory Behavior of the SPEC2000 Benchmark Suit," Tech. Report, IBM Corp. Oct. 2000.

[21] Y. Sazeides and J. Smith, "The Predictability of Data Values," Proc. of 30th Int'l Symp. on Microarchitecture, Triangle Park, NC, Dec. 1997, pp. 248-258.

[22] S.T. Srinivasan, R.D. Ju, A.R. Lebeck and C. Wilkerson, "Locality vs. Criticality," Proc. of 28th Int'l Symp. on Computer Architecture, Goteborg, Sweden, July 2001, pp.132-143.

[23] G. Tyson and T. Austin, "Improving the Accuracy and Performance of Memory Communication Through Renaming," Proc. of 30th Int'l Symp. on Microarchitecture, Triangle Park, NC, Dec. 1997, pp. 218-227.

[24] K. Wang, and M. Franklin, "Highly Accurate Data Value Prediction using Hybrid Predictors," Proc. of 30th Int'l Symp. on Microarchitecture, Triangle Park, NC, Dec. 1997, pp. 281-290.

[25] K. M. Wilson, K. Olukoton and M. Rosenblum, "Increasing cache port efficiency for dynamic superScalar microprocessors," Proc. of 23rd Int'l Symp. on Computer Architecture, Philadelphia, PA, May 1996, pp.147-157.