Address-Free Memory Access Based on Program Syntax Correlation of Loads and Stores

Lu Peng, Jih-Kwon Peir, Qianrong Ma, and Konrad Lai

***Abstract*—An increasing cache latency in next-generation processors incurs profound performance impacts in spite of advanced out-of-order execution techniques. One way to circumvent this cache latency problem is to predict load values at the onset of pipeline execution by exploiting either the load value locality or the address correlation of stores and loads. In this paper, we describe a new load value speculation mechanism based on the program syntax correlation of stores and loads. We establish a *symbolic cache (SC)*, which is accessed in early pipeline stages to achieve a zero-cycle load. Instead of using memory addresses, the SC is accessed by the encoding bits of base register ID plus the displacement directly from the instruction code. Performance evaluations using SPEC95 and SPEC2000 integer programs on SimpleScalar simulation tools show that the SC achieves higher prediction accuracy in comparison with other load value speculation methods, especially when hardware resources are limited.**

I. Introduction

TODAY'S high-performance processor pipeline permits overlapping instruction execution to achieve more than one instruction per cycle (IPC) average execution rate. The available instruction-level parallelism (ILP) constrains this parallel execution because dependent instructions must wait for the data produced by the source instructions. The severity, in terms of execution delays, depends primarily on the speed that the producer instruction can generate the needed data.

Memory load latency presents a classical pipeline bottleneck even when the data is located in the first-level cache (L_1). Usually, the load data from L_1 is not ready until late stages of the pipeline while the dependent instruction requires the data at an earlier stage. This load-to-use delay exacerbates in recent high-performance microprocessors in which multicycle, first-level caches become the norm [12], [14], [21], [23], [24]. As the cache size, clock frequency, and complexity of microarchitecture continue to increase in next-generation processors, it is estimated that the L_1 cache accesses may consume two to five cycles [2]. This increasing load latency from caches will further lengthen the load-to-use delay and will have profound performance impacts in spite of advanced out-of-order execution techniques [2], [3], [18]. Simulations using SPEC2000 integer benchmarks running on the out-of-order SimpleScalar model [4] have shown that each cycle reduction of the L_1 access delay improves the IPC by 5%–10% [18].

In Fig. 1, a conceptual out-of-order execution pipeline is partitioned into two phases. First, an instruction is fetched, decoded, renamed, and issued through the *front end* of pipeline stages. Afterwards, the register operands are read and the instruction is executed (including memory access) and committed through the *back end* of pipeline stages. In order to be stall free, a source instruction must produce the data before its dependent executions. In other words, a critical producer, when it is fetched and issued at the same cycle as its dependent instructions, needs to generate the result in the front end of the pipeline to avoid any stall of its dependents. Such a dependent stall-free memory load instruction is called a *zero-cycle* load.

There have been several attempts to achieve a zero-cycle load by predicting and speculating the load value [5], [11], [15], [16], [22], [25], [26] or the load address [2], [6], [9], [10] in the front-end of the processor pipeline. Both load value and load address predictions generally suffer a low prediction accuracy. For address predictions, a lengthy cache access is still required that may delay the load dependents even if the predicted load address is correct.

In this paper, we exploit a new avenue to speculatively obtain the load value in front-end stages of the pipeline. First, we observe that store-load and load-load correlations are established in software and often displayed in the program syntax in the form of a base register ID plus a displacement value. Therefore, it is reasonable to use part of the store-load encoding bits (base register ID $+$ displacement) directly to capture such correlations. Second, applications exhibit spatial locality among memory references. Such locality can also be observed in the program syntax when nearby loads or stores differ only by a small displacement value. Therefore, it is beneficial to establish store-load dependences on a large block granularity to capture the spatial data reference locality.

The syntax correlation holds when the content of the base register remains unchanged. This property exists in various program constructs such as accessing global and local variables, saving/restoring registers during procedure/function calls, referencing different records using the same pointer in linked data structures, accessing array elements in loop iterations with/without loop unrolling, etc. We also observe that the base address may stay the same even when the base register is updated between two memory references. This is due to a lack of sufficient registers, an uncertainty of future execution paths, or a traversal through different procedures that requires a base register to be saved and restored before the next usage.

Manuscript received June 9, 2002; revised September 10, 2002. This work was supported in part by National Science Foundation Grants MIP-9624498, EIA-0073473, and in part by Intel Research and Equipment Donations.

L. Peng and J.-K. Peir are with the Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL 32611 USA (e-mail: lpeng@cise.ufl.edu; peir@cise.ufl.edu).

Q. Ma is with Server Technology, Oracle Corporation, Redwood City, CA 94065 USA (e-mail: qianrong.ma@oracle.com).

K. Lai is with Microprocessor Research, Intel Lab, Intel Corporation, Hillsboro, OR 97124 USA (e-mail: konrad.lai@intel.com).

Digital Object Identifier 10.1109/TVLSI.2003.812315

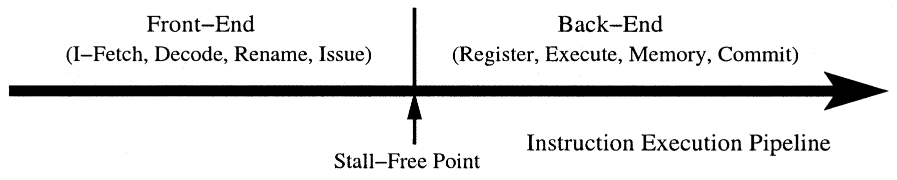

Front–End
(I–Fetch, Decode, Rename, Issue)
Back–End
(Register, Execute, Memory, Commit)
Stall–Free Point
Instruction Execution Pipeline

Fig. 1. Processor pipeline and dependent stall-free point.

Based on these observations, we propose a *symbolic cache (SC)*. An SC is a small separate data cache that is accessed at early front-end stages using certain encoding bits directly from memory instructions. The speculative data retrieved from the SC can trigger the execution of dependent instructions to avoid any delays. Performance evaluations using SimpleScalar tools and SPEC95/SPEC2000 integer programs show that the average prediction accuracy reaches over 70% using small SCs. This accuracy is generally higher than other data speculation methods, especially when hardware resources are limited for constructing extra caches and tables. The remaining paper is organized as follows. A few related work on hiding cache latency will be given in the next section. The motivations and important observations for the proposed method will be described in Section III. This is followed by discussions of design and related issues for establishing the SC in Section IV. In Section V, performance evaluations of three-data speculation methods are given. Several design parameters for the SC are also evaluated. Finally, Section VI concludes the paper.

II. Related Work

The most aggressive load value speculation is to predict the value at the onset of pipeline execution. A load-value history table is established and accessed using the program counter (PC) of the load. This scheme allows loads bypassing caches completely to achieve a zero-cycle load. A value prediction can be successful if the value is repeated from the previous execution of the load [15], [16], [26], or the load value is followed certain recurrence patterns [22]. However, the lack of a close correlation between the instruction address and the value of the load makes it difficult to achieve a high prediction accuracy [5], [11], [15], [16], [22], [26].

Another way to circumvent pipeline hazards caused by the cache latency is to predict the load address at the onset of pipeline execution so that a cache access can start speculatively without going through the normal decode, rename, and address generation stages [2], [6], [9], [10]. Existing address prediction methods exploit regular patterns such as stride-based address patterns, and irregular but repeated patterns such as addresses for traversing link-based data structure. However, the difficulty remains of predicting a significant portion (over 30% [2]) of load addresses that do not fall into these two categories. In a recent proposal, dynamic dependence links were established between the instruction which updates a register to the instruction where the register is used as the base register [8]. Once the updated value is available, the dependent load address can be calculated early and more accurately. However, the lengthy cache access is required still, even with a correct address.

Memory renaming techniques establish dynamic dependence correlations between stores and loads [25]. A separate storage element called a *value file (VF)* 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 a lengthy cache access. Studies show that there are many more loads that consume the value from the same producer than those loads which repeat the same value or address from the previous instance of the same load. Therefore, there is a better chance to obtain the correct load value by using memory renaming through the VF rather than based on the load value/address locality. This approach, however, requires additional hardware to establish the correct dependence links among stores and loads. The load value cannot be accurately predicted before such a correlation has established dynamically. A similar idea has been exploited to dynamically establish store-load [19] and load-load [20] *associations*. A small *synonym file* which keeps the correlated data can be indirected accessed by the PC of the load.

Recently, another early load address resolution technique for deep-pipelined machines has been proposed [3]. The authors observed that the addresses for certain types of memory loads, such as stack access, constant, or stride-based memory access, have regular increment/decrement patterns. By tracking the registers used for this type of load, register updates can be computed at the decode stage. As a consequence, the dependent load can start the address generation and cache access earlier after the load is decoded. Although nonspeculative, this approach is limited to memory loads with certain address patterns. Also, the lengthy cache access is still required.

There have been other attempts to achieve fast cache accesses. The real cache index bit prediction based on the base register content enables parallel address translation and cache access [13]. Due to small offset values, the zero-cycle load technique [1] uses a simple carry-free adder for fast approximation of the load address. To avoid speculative address calculations, a special compiler-directed register is added in [7] to save the content of the base register for the next load so that the load address can be calculated in the decode stage. The SAM cache [17] uses the base address and the offset separately to access the cache directly. Although all these techniques achieve fast cache access, their impact in hiding the long cache latency on deep-pipelined microarchitectures is rather limited.

The proposed SC has several advantages over existing cache latency hiding methods. First, the SC can handle any type of loads, address patterns, or special usages of base registers. Second, unlike address predictions or register tracking, loads through the SC can bypass the address generation and cache access completely to achieve a zero-cycle load. This is similar

```
Disjoint * 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**

Fig. 2. Example I: source and assembly codes of function copy-disjunct from Parser.

to the value prediction method. However, instead of being based upon the history of the load values, the SC captures store/load syntax correlations with higher accuracy. Third, unlike the memory renaming technique, where the store/load correlation is established dynamically by the hardware, the store/load correlation is directly obtained from the instruction encoding bits to simplify the hardware requirement. In addition, the SC can capture spatial locality among memory references.

III. Syntax Correlation of Memory References

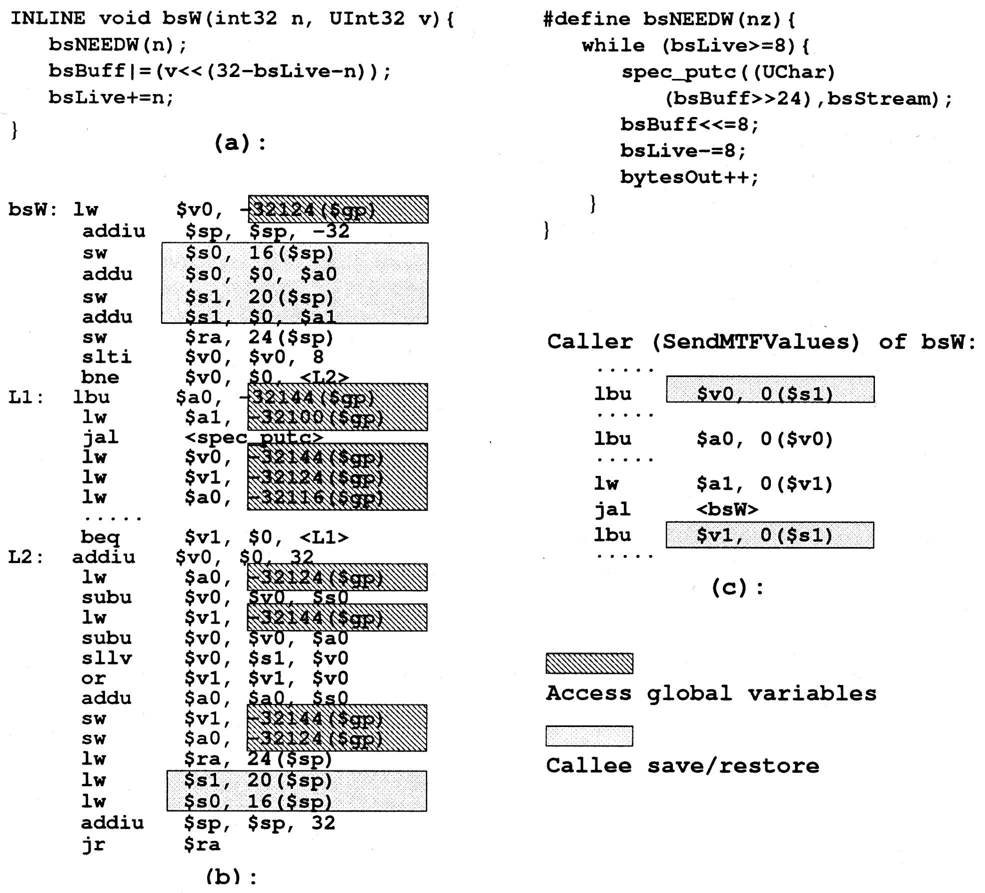
The foundation of the SC is based on store-load and load-load correlations from the program syntax in the form of a base register ID and a displacement value. This simple memory reference syntax also exhibits spatial locality. In this section, we will provide two programming examples and describe qualitatively the existence of such syntax correlations and reference locality in real programs. In Fig. 2, the source and the assembly codes of a simple function *copy_disjunct from Parser* of SPEC2000 are given. This function is invoked many times to build a new copy of a disjunct list. The second example bsW is extracted from *Bzip* of SPEC2000 (Fig. 3). This function is also invoked multiple times to perform bit-stream I/Os.

The store/load syntax correlation and reference locality can be observed in several program constructs.

Register Save and Restore in Procedures and Functions: As shown in Fig. 2, store/load dependences can be established perfectly with a matching pair of the base register ($\$sp$) and displacement for saving and restoring register contents when the function *copy_disjunct* is invoked. Although the invocations of xalloc and *copy_connectors* may change the value of the $\$sp$, the original value in the *copy_disjunct* is restored after returning from the function calls.

Access Records in Linked Data Structures: In the same example, the pointers (d, $d1$) are used to copy and construct a new node in the target 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 records with small variations of the displacement value. The syntax correlations and reference locality among these accesses are clearly demonstrated in the assembly code.

Access Array Variables: Similar store/load correlations are also observed in accessing array data structures in several



```
INLINE void bsW(int32 n, UInt32 v){
   bsNEEDW(n);
   bsBuff|=(v<<(32-bsLive-n));
   bsLive+=n;
}
```
(a):

```
#define bsNEEDW(nz){
   while (bsLive>=8){
      spec_putc((UChar)
         (bsBuff>>24),bsStream);
      bsBuff<<=8;
      bsLive-=8;
      bytesOut++;
   }
}
```

```
bsW: lw      $v0, -32124($gp)
     addiu   $sp, $sp, -32
     sw      $s0, 16($sp)
     addu    $s0, $0, $a0
     sw      $s1, 20($sp)
     addu    $s1, $0, $a1
     sw      $ra, 24($sp)
     slti    $v0, $v0, 8
     bne     $v0, $0, <L2>
L1:  lbu     $a0, -32144($gp)
     lw      $a1, -32100($gp)
     jal     <spec_putc>
     lw      $v0, -32144($gp)
     lw      $v1, -32124($gp)
     lw      $a0, -32116($gp)
     .....
     beq     $v1, $0, <L1>
L2:  addiu   $v0, $0, 32
     lw      $a0, -32124($gp)
     subu    $v0, $v0, $s0
     lw      $v1, -32144($gp)
     subu    $v0, $v0, $a0
     sllv    $v0, $s1, $v0
     or      $v1, $v1, $v0
     addu    $a0, $a0, $s0
     sw      $v1, -32144($gp)
     sw      $a0, -32124($gp)
     lw      $ra, 24($sp)
     lw      $s1, 20($sp)
     lw      $s0, 16($sp)
     addiu   $sp, $sp, 32
     jr      $ra
```
(b):

```
Caller (SendMTFValues) of bsW:
     .....
     lbu     $v0, 0($s1)
     .....
     lbu     $a0, 0($v0)
     .....
     lw      $a1, 0($v1)
     jal     <bsW>
     lbu     $v1, 0($s1)
     .....
```
(c):

▨ Access global variables

░ Callee save/restore

Fig. 3. Example II: function bsW from Bzip. (a) Source code. (b) Assembly code. (c) Partial assembly code from caller SendMTFValues.

studied workload. For example, intensive array accesses are observed in several functions in *Gcc* of SPEC2000. Nearby references to different elements of the same array with the same base address provide syntax correlated stores and loads.

Access Global Variables: As shown in Fig. 3, 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 $\$gp$. The access of global variables exhibits both the syntax correlation and the 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 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 $\$s8$ that also display strong syntax correlations and spatial locality.

Save/Restore Base Registers: There are evidences that the syntax correlation is still hold even if the base register has been updated between two memory accesses. This is due mainly to the fact that a base register may be freed up for other usages and the original base address is restored before the next memory reference. In Fig. 3, we also show 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, $\$s1$ continues to be used as a base register. Although $\$s1$ has been updated in the *bsW*, the original base address is restored to keep the syntax correlation alive.

IV. Establishing a Symbolic Cache

An SC is a small data cache which is addressed by the encoding content of load/store instructions. The SC can be accessed once loads/stores are fetched out of the instruction cache. As a result, pipeline stages involving register file access, address generation/translation, and cache access can be bypassed. The impact of pipeline performance using an SC is very similar to that of using the VF in memory renaming techniques [25], where the speculative load data is fetched out of the VF indirectly through a store/load correlation table. In this paper, we focus on the accuracy of load data speculation using the SC. We omit discussions of integrating the SC into a pipeline microarchitecture.

It is essential to properly extract the symbolic address from the encoding bits of load/store instructions to capture the syntax correlations. A typical memory instruction consists of an opcode, a register source/destination, and a memory source/destination. Intuitively, we can use the memory source/destination to form a 32-bit symbolic address as illustrated in Fig. 4. The least significant 16 bits are extracted from the displacement value, and the base register ID (5 bits) are inserted next to the displacement. Although simple, this approach suffers aliasing problems because multiple memory addresses can be mapped to the same

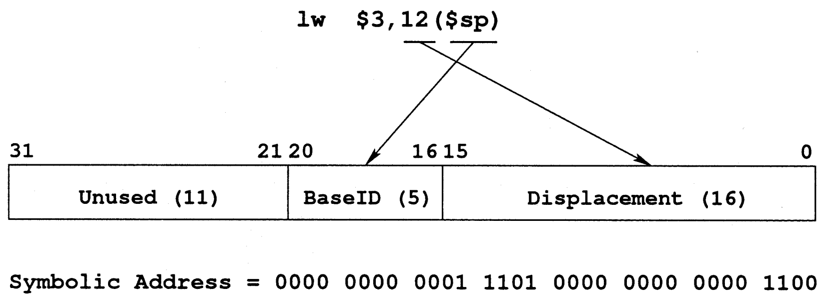


Fig. 4. Extracting symbolic address from memory instructions.

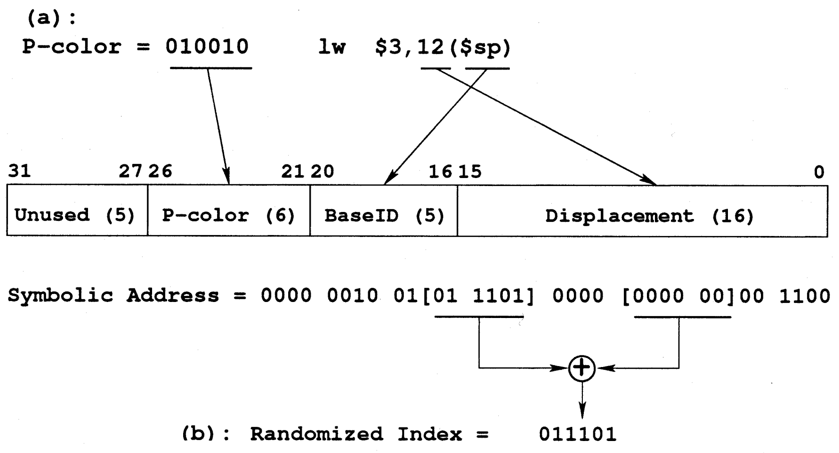


Fig. 5. (a) Adding procedure color to symbolic address. (b) Index randomization in accessing the SC.

symbolic address. In addition, this simple symbolic address formation creates other access and alignment problems.

- *Aliasing of Symbolic Address*: With the simple address mapping in Fig. 4, a 32-bit memory address is represented by a 21-bit symbolic address. Therefore, multiple memory addresses can be expressed by the same symbolic address. An obvious example can be found in stack accesses for local variables and for saving and restoring registers during procedure/function calls. Although accessing a different stack frame in each procedure invocation, the same stack pointer ($\$sp$) and frame pointer ($\$s8$) with a small range of displacement values are commonly used. The contents in the SC for local variables and saved registers are likely overwritten in the callee procedures and cannot be reused after returning from the procedures.
- *Uneven SC Index Distribution*: It is well known that displacement values in memory references are unevenly distributed with a high percentage of “0” and a few other constants. Using a portion of the high-order displacement bits as the index to the SC may potentially generate heavy conflict misses.
- *Word/Byte Alignment*: The most difficult problem lies in the difference of the line boundary between a symbolic and a L_1 cache lines. This alignment problem is due to the fact that offset bits of a cache line are not always the same between the symbolic and the real addresses. It is essential to properly align the data layout in the symbolic cache according to the symbolic address to capture the spatial locality of memory references.

A. Procedure Coloring and Index Randomization

In order to alleviate the stack access aliasing problem in different procedures, various procedure coloring techniques can be constructed. A straight-forward technique is to maintain a global counter called *P-color*. The P-color is incremented whenever a procedure call is encountered. It is decremented after returning from a procedure. The P-color can be incremented contiguously in nested or recursive procedures before being decremented. Stack accesses between a caller and its callees can be differentiated by the P-color to avoid conflicts in the SC.

The P-color can be concatenated with the symbolic address for stack accesses. The width of the P-color counter is flexible. Fig. 5(a) illustrates the symbolic address after adding a 6-bit P-color. It is important to know that the P-color is only applied to stack accesses which use $\$sp$ and $\$s8$ as the base register. Other memory accesses do not add the P-color to allow sharing of global variables among different procedures or functions.

An uneven distribution of the index bits extracted directly from the displacement value has a potential to create heavy conflict misses in the SC. This problem comes from the fact that high-order displacement bits are often all zeros and can be dealt with by a simple randomization technique. Instead of extracting index bits from the symbolic address directly, randomized index bits can be formed by *exclusive-ORing* the original index bits from the displacement with the bits from the base register ID and the P-color as illustrated in Fig. 5(b). In this example, it is assumed that the SC has 64 sets with 64-B line size. The six

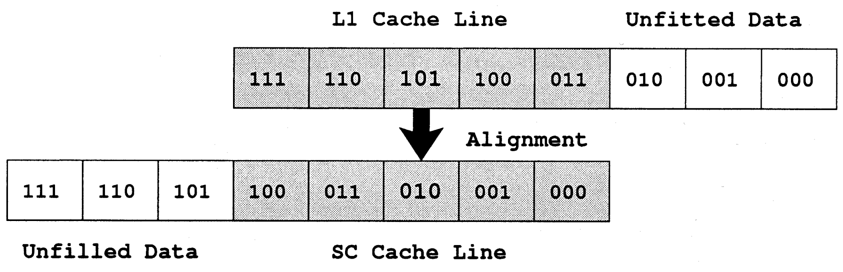


Fig. 6. Data alignment in symbolic cache.

index bits are obtained by *exclusive-ORing* normal index bits in position 6 to 11 with the base register ID and partial P-color bits starting at position 16–21.

B. Word/Byte Alignment

One remaining issue is the data alignment between the SC and the L_1 data cache. The symbolic address within a cache line, i.e. the last few offset bits, may not be the same as the offset bits in the real address. In order to exploit spatial reference locality, the cache line fetched from L_1 needs to be rearranged in the SC such that the data layout can be aligned with the symbolic address. The basic alignment algorithm works as follows. When a memory request misses the SC, the target cache line is fetched from the memory hierarchy and loaded into the SC. The target byte/word is placed in the SC according to offset bits of the symbolic address. For example, assume there are eight access units in a cache line as shown in Fig. 6. The symbolic offset of the target unit is 010 while the offset of the real address is 101. In this case, the target data 101 is loaded into unit 010 in the SC. The remaining units are loaded according to the location of the target unit. There are thus two important aspects to consider for a proper data alignment.

1) *Granularity of Data Alignment*: Depending on memory access granularity, it is conceivable that the data alignment can be performed at byte, half-word, word, or double-word level. The byte-level alignment can accommodate accesses by other granularity with the expense of maintaining more valid bits for the alignment information.
2) *Handling Underflow/Overflow Data*: Since the line boundaries of the SC and the L_1 caches may be different, only a partial line can be filled on each SC miss. In addition, there is excessive data from the target L_1 cache line that cannot fit into the requested line location in the SC. The simplest and most natural solution is to only fill a partial SC line and drop the unfitted data. Other options include fetching two adjacent L_1 lines for each requested SC line, and/or to search and place the overflow L_1 data into the correct second SC line.

Performance evaluation on these design options will be given in the next section. It is important to keep the SC design simple since the primary goal of establishing the SC is to provide a zero-cycle load.

V. Performance Evaluation

Performance evaluations of three load value speculation methods are given including the last value and stride-based value prediction (VP), the memory renaming (MR), and the proposed symbolic cache (SC). Our primary focus is to compare the prediction accuracy among these three mechanisms. All simulations are carried out on the *Sim-Save* model of SimpleScalar. Twelve integer programs, *Go, Li, M88k, Perl* from SPEC95 and *Bzip, Gcc, Gzip, Mcf, Parser, Twolf, Vortex, Vpr* from SPEC2000 are used. Version 2.7.2.3 ssbig-na-sstrix-gcc compiler with options: (-funroll-loops -O2) is used to generate the binary code. For each workload, we skip the first 900 million instructions, use the next 100 million instructions to warm up the caches and tables, then collect simulation statistics from the next 500 million instructions.

A. Data Alignment

We first investigate and evaluate different alignment granularity. Table I shows matches of the least-significant two bits between the symbolic and the real addresses with different memory access granularity in the simulated programs. On the average, 87.4%, 3.2%, and 9.4% of memory references are accessing word, half-word, and byte respectively. Mismatches of the two bits for the three access granularities are about 0%, 0.5%, and 4.5%. The word access is always aligned at the word boundary for both the real and the symbolic addresses. On the other hand, the word alignment creates 5% of mismatches for half-word and byte accesses. Since the word alignment reduces extra valid bits significantly, we will simulate both byte and word alignments and show their impact on the SC accuracy.

With regards to the line fill on SC misses, preliminary studies show that the option of filling the entire SC line by fetching potentially more than one L_1 cache lines provides very limited benefit. Moreover, to place the entire target L_1 line into the SC on each miss does not benefit the accuracy much either. Therefore, only the simple partial SC line fill by dropping any unfitted data is considered in subsequent evaluations.

B. Sensitivity of P-Color and Index Randomization

Table II shows the accuracy of load-value speculation using a 4 KB SC with the word-alignment and 0, 2, 4, P-color bits. In general, we observe an average improvement from 68.8% to 70.4% by adding a 2-bit P-color. A few benchmark programs show no improvement at all with the simple P-color mechanism. After examining dynamic function calls in these programs, we found that there are very few nested calls and the program ex-

TABLE I
MATCHING OF THE TWO LEAST-SIGNIFICANT ADDRESS BITS BETWEEN REAL AND SYMBOLIC ADDRESSES FOR ACCESSING WORD, HALF-WORD, AND BYTE

	Word		Hword		Byte		Total	
	match	unmatch	match	unmatch	match	unmatch	match	unmatch
Go	100.0	0.0	0.0	0.0	0.0	0.0	100.0	0.0
Li	88.7	0.0	0.0	0.0	9.7	1.6	98.4	1.6
M88k	93.1	0.0	0.3	0.0	6.6	0.0	100.0	0.0
Perl	87.6	0.0	0.0	0.0	11.7	0.8	99.2	0.8
Bzip	60.1	0.0	5.0	2.6	9.6	22.8	74.6	25.4
Gcc	91.9	0.0	5.8	0.1	1.4	0.9	99.1	0.9
Gzip	60.7	0.0	13.7	3.0	10.4	12.3	84.8	15.2
Mcf	100.0	0.0	0.0	0.0	0.0	0.0	100.0	0.0
Parser	99.2	0.0	0.0	0.0	0.2	0.5	99.5	0.5
Twolf	71.0	0.0	5.0	0.3	9.5	14.2	85.5	14.5
Vortex	96.7	0.0	2.3	0.0	0.4	0.6	99.4	0.6
Vpr	100.0	0.0	0.0	0.0	0.0	0.0	100.0	0.0
Average	87.4	0.0	2.7	0.5	4.9	4.5	95.0	5.0

TABLE II
LOAD ACCURACY USING THE SC WITH/WITHOUT THE P-COLOR

	Go	Li	M88ksim	Perl	Bzip	Gcc	Gzip	Mcf	Parser	Twolf	Vortex	Vpr	Average
No-color	45.9	72.4	90.0	75.4	51.9	83.9	66.4	62.2	62.6	74.4	70.7	70.1	68.8
2-bit	46.3	73.8	91.5	79.6	52.9	85.3	66.4	62.4	65.8	74.0	74.5	72.5	70.4
4-bit	46.3	73.8	91.5	79.6	52.9	85.3	66.4	62.4	65.8	74.0	74.5	72.5	70.4

TABLE III
LOAD ACCURACY USING THE SC WITH INDEX RANDOMIZATION

	Go	Li	M88k	Perl	Bzip	Gcc	Gzip	Mcf	Parser	Twolf	Vortex	Vpr	Average
4-way	43.0	69.3	86.2	71.2	50.1	81.9	66.4	57.2	56.4	70.8	63.6	62.8	64.9
Fully-asso	47.5	76.9	92.3	80.0	52.9	85.6	66.4	62.5	68.5	74.2	76.4	72.5	71.3
Random	46.3	73.8	91.5	79.6	52.9	85.3	66.4	62.4	65.8	74.0	74.5	72.5	70.4

ecution tends not to frequently traverse back and forth among multiple procedure levels. For instance, in Gzip, about 98% of the calls are labeled at level 6. We also observe that there is no benefit in increasing the number of bits in the P-color. With more P-colors, more levels of procedure invocation can be differentiated. However, analysis of application programs reveals that perfectly-nested or deeply-recursive procedures that benefit with more P-colors rarely exist. The actual execution path normally traverses among a few levels of procedures. Also, due to a small SC, the data from ancient ancestors is difficult to hold anyway.

The benefit of index randomization is more evident in Table III, in which the accuracies of three 4 KB SC configurations are displayed. By randomizing the index, a 4-way set-associative SC can achieve the accuracy approaching to that of a fully-associative SC. On the other hand, without this process, it degrades the accuracy of the 4-way design from 70.4% to 64.9%.

These results suggest that the effective working set between base register updates is very small. Once the content of a base register changes, the old data in SC based on the same base register becomes stale. Because the original index bits are likely to be all zeros (Fig. 5), stores and loads using the same base register may locate in very few sets even with index randomizations. Given the fact that the randomized 4-way SC achieves an accuracy comparable to that of a fully-associative SC, the four lines in each set are enough to hold the working set for each base register ID. Although higher set associativities increase the capacity in each set to hold more lines for each base register, frequent updates of base registers wipe out the corresponding correlated data in the SC.

C. Comparison of Three-Data Speculation Methods

The accuracies of three load value speculation mechanisms are evaluated. Both byte and word alignments for placing a line in the SC are considered. Also, index randomizations and a 2-bit P-color are applied to improve the load accuracy. For a fair comparison, we simulate the three methods using comparable hardware with respect to the extra storage requirement to build additional tables and caches.

The VP scheme establishes a value history table to remember the recent value of each load. For matching the PC of a load, proper tags are maintained in the value history table. In addition, an increment value is needed in each entry to accommodate a stride-based predictor. The MR scheme uses a VF to keep store/load correlated values for later accesses. In addition, two extra tables are needed. The store/load cache (SLC) saves pointers to the VF. The SLC is addressed by the PCs of loads and stores with tags for matching the correct PC for indirect accesses to the VF. The store-address cache (SAC) also records pointers to the VF. The SAC is accessed by load/store addresses for establishing load/store correlations. Again, address tags are necessary to make a correct correlation. The SC is simply a data

TABLE IV
SIX CONFIGURATIONS FOR ACCURACY COMPARISONS

Configurations	VP (word)	MR VF (word)	MR SLC (link)	MR SAC (link)	SC (line/size)
1	128	64	128	128	16/1KB
2	256	128	256	256	32/2KB
3	512	256	512	512	64/4KB
4	1024	512	1024	1024	128/8KB
5	2048	1024	2048	2048	256/16KB
6	4096	2048	4096	4096	512/32KB

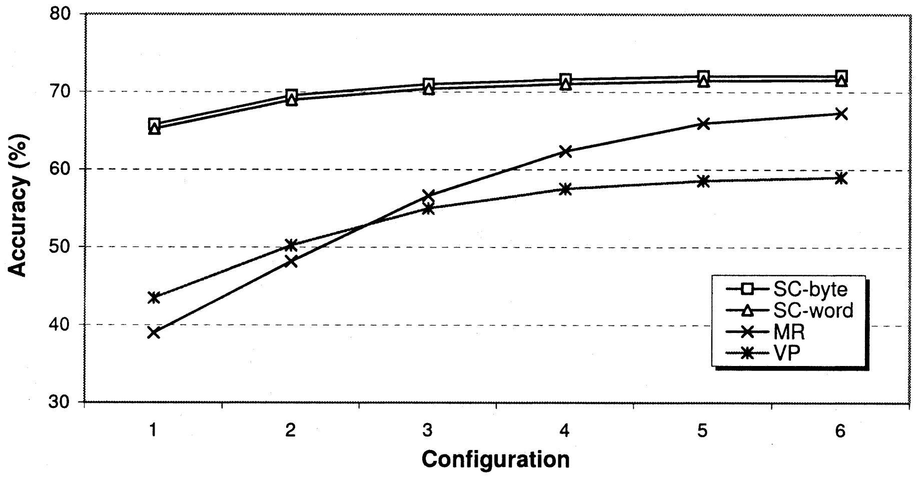


Fig. 7. Average accuracies of three-data speculation methods.

cache addressed by the symbolic address. There is no extra hardware except for a small tag array in which each tag along with a few valid bits is associated with a 64-B symbolic cache line.

We consider six configurations for accuracy comparisons as shown in Table IV. The hardware requirement is represented by the total number of entries in the respective tables and caches. Because of the additional tag arrays, the storage requirement for the VP and the MR are actually about 40%–50% and 10%–15% more than that of the SC in each configuration. Note that in this first-cut estimation, extra control logic is not considered.

Fig. 7 plots the average accuracy curves based on the twelve integer programs for the three-data speculation methods. Generally speaking, the SC has the highest accuracy, especially with small configurations. For example, more than 70% of the loads can obtained correct values from a small 4 KB SC. These results demonstrate the existence of store/load syntax correlations and spatial locality that can be captured effectively by small SCs. The MR scheme, on the other hand, requires eight times of the hardware storage to reach about 67% accuracy. The MR scheme performs poorly with small configurations primarily because of misses to the small SLC/SAC for establishing correct store/load correlations. In addition, the correlation must be established before a correct value can be obtained. The MR scheme shows more improvement when the configuration size increases. With bigger SLC/SAC, data dependence links can be built more precisely than those approximated by the symbolic address. However, the SC still maintains an edge by capturing the spatial locality. The last/stride value predictor generally has the worst accuracy. The accuracy improvement is leveling off with larger value history tables. This confirms a poor correlation between the load value and its instruction address.

The byte alignment does not improve the accuracy much. For a 4 KB SC, for instance, the byte alignment improves the average accuracy of the word alignment from 70.4%–71.1%. As shown in Table I, there is very little or no difference between byte or word alignment for a majority of the programs. The two programs that benefit the byte alignment the most are *Bzip* and *Gzip* because of their high percentage of subword accesses and mismatches of the least-significant 2 bits between real and symbolic addresses.

The SC size plays a minor role in providing accurate load values. Again, this is due to the fact that the working set between base register updates is very small. Since the randomized SC index is still mapped to very few sets for each base register, increasing the SC size (i.e. the number of sets) does not improve the capacity for loads using a specific base register.

Now considering the third configuration with a 4 KB SC, the average prediction accuracies are 55.0%, 56.6%, 70.4%, and 71.1% for the VP, the MR, and the SC with word alignment (SC-word) and the SC with byte alignment (SC-byte), respectively, as shown in Fig. 8. Among the twelve integer programs, *M88k*, *Perl*, and *Gcc* show very good syntax correlations with over 80% of prediction accuracies, while *Li*, *Gzip*, *Twolf*, *Vortex*, and *Vpr* show reasonable accuracies over 70%. *Go*, *Bzip*, *Mcf*, and *Parser*, on the other hand, have poor accuracy, especially for *Go* with an accuracy only about 47%. Recall that in order

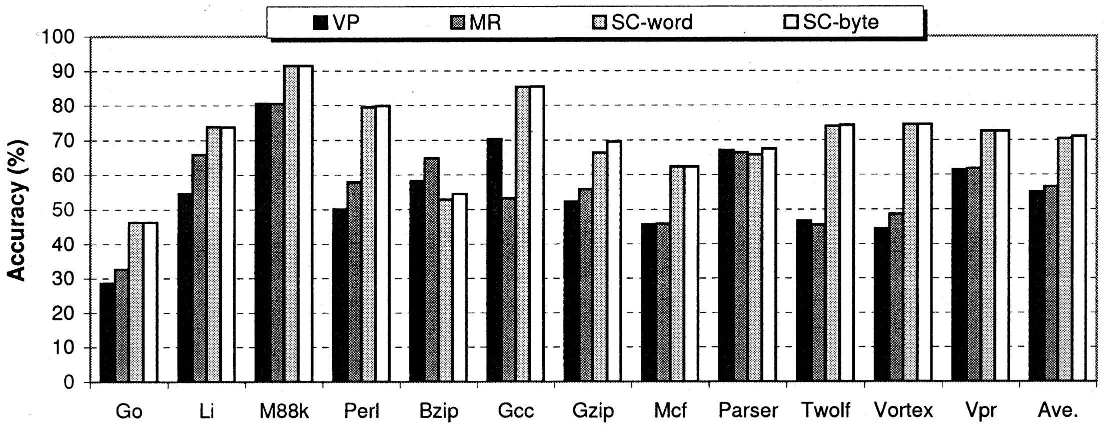


Fig. 8. Accuracy of three-data speculation methods for individual programs (based on configuration 3).

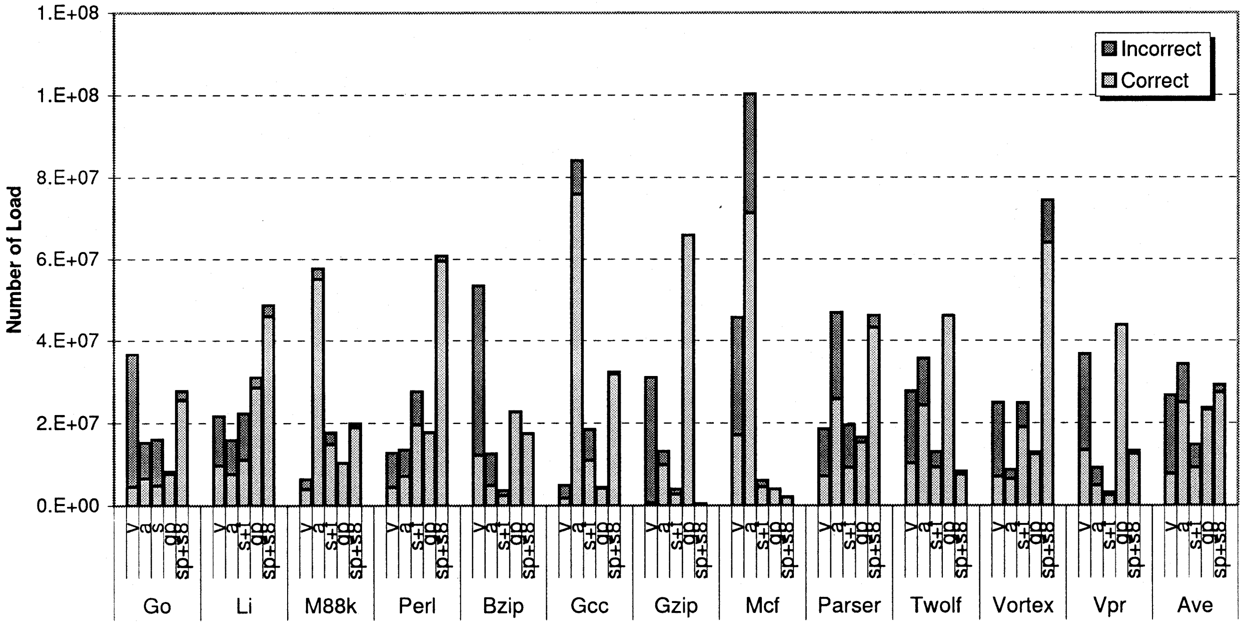


Fig. 9. Correct/incorrect load value speculations with respect to different base register groups.

to hold the syntax correlation, the base register content must remain unchanged between two correlative memory instructions. We found out in *Go*, about 64% of the loads are executed using a newly updated base register. On the other hand, only 22% and 24%, respectively, for the loads in *Gcc* and *M88k* are executed right after their base registers have updated. More detailed analysis with respect to the base register updates will be given in the next Section V-D.

The SC scheme does not perform well against the other two schemes under *Bzip* and *Parser*. In *Bzip*, a main function *fullGtU* that finds matches of character strings, has shown good value locality and good dynamic store/load correlations established by the MR scheme. However, the SC handles this function poorly because the base addresses of the matching strings are calculated right before loading characters from the two strings. A similar behavior has also found in *Parser*.

In Fig. 9, we break down correct and incorrect load value speculations using the SC with respect to the base register IDs. We separate base registers into five groups: $\$v$, $\$a$, $\$s + \$t$, $\$gp$ and $\$sp + \$s8$, each represents 20.5%, 26.4%, 11.3%, 18.2%, and 22.5% of the total loads, respectively. (Note there is about 1% of the loads using other registers.) The accuracies of the five base register groups are 29%, 73%, 63%, 98%, and 94%. As expected, it is highly 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 computed or loaded from memory for an indirect access right before the load that results in an incorrect values from the SC. The $\$a$ registers, which show higher accuracy, are also used for passing parameters to callee functions. We observe that many functions have memory addresses (pointers) as parameters that are passing through the $\$a$ registers. In each callee function, the $\$a$ registers are frequently used as a base without any modification. We also found in Gcc that certain memory addresses are passing through several function levels using the $\$a$ registers. Thus, memory loads based on $\$a$ can potentially keep the correlations alive through several function levels.

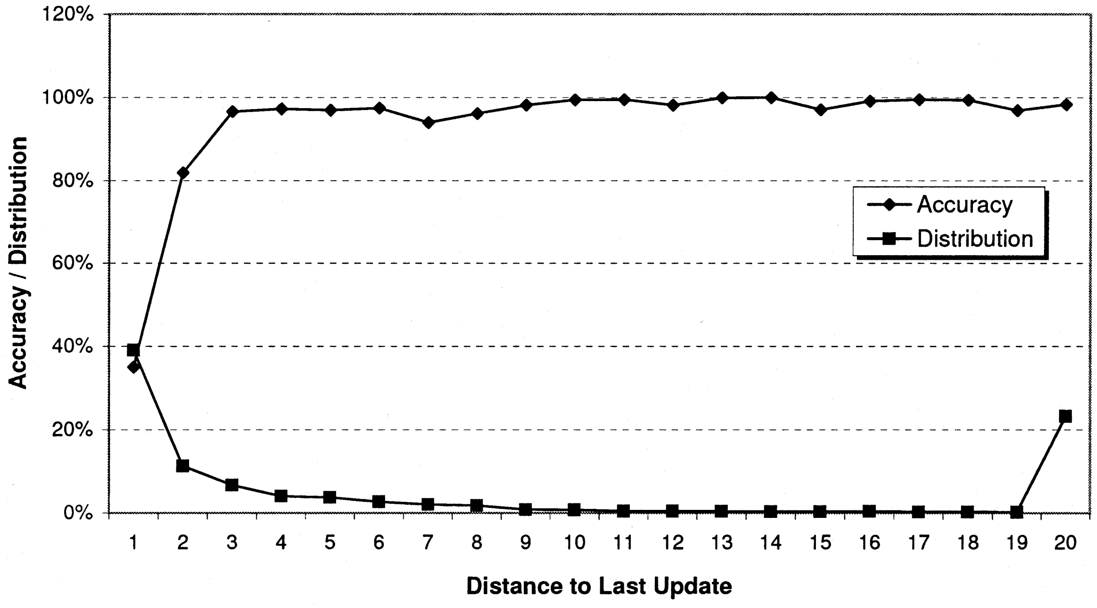


Fig. 10. Load accuracy and distribution with respect to the distance to the last base register update.

D. Accuracy Regarding Base Register Update

The syntax correlation holds when the content of the base register remains unchanged from the last memory reference with the same symbolic address. Fig. 10 shows the average accuracy of all the loads with respect to the distance to the last update of the base register. For example, the distance is equal to 1 for a load when the base register of the load is used for the first time as a base register after an update to the register. Similarly, the distance is equal to 2 if a register is used for the second time as a base register for either load or store after the content of the register has updated. The distances of 20 or longer are represented by a single data point. In general, the accuracy goes up with the distance due to the locality of references. A cold miss is encountered when the distance is equal to 1 unless the latest update did not change the content of the base register from the previous use of the same base register.

A few observations can be made from the figure. First, when the distance is 3 or longer, the speculative load data from the SC is very accurate with an average accuracy about 98%. This indicates a very strong reference locality based on the symbolic addresses of nearby stores and loads.

Second, instead of all cold misses, the average accuracy is 36% when the distance is equal to 1. This accuracy comes from restoring base register content before the load. Unfortunately, a significant portion (39%) of the loads use a base register at the first time after its updates. With only 36% of accuracy, these loads produce 25% inaccurate data with respect to the total loads. Therefore, the distance-1 loads are the major factor for the overall accuracy. For example, in the two high-accuracy programs, *Gcc* and M88k, only 22% and 24% of loads are distance-1 with an accuracy of 46% and 68%, respectively. On the other hand, *Go* has 64% of loads are distance-1 with a poor accuracy of 18%.

Third, about 24% of the loads have distances of 20 or longer. This long distance comes mainly from access global variables, also from some local variable accesses. An average accuracy of 98.4% is obtained for these long-distance loads.

Compiler optimization techniques may be applied to improve the syntax correlations of stores/loads. For example, we observe that parameters are sometimes passed to the callee through the caller's stack frame. Accessing the parameters before an update of the frame pointer may keep the correlation alive. Further discussions in this direction is out of the scope of this paper.

VI. Conclusion

A new load data-speculation method, based on instruction syntax correlations of stores and loads, has been introduced in this paper. Instead of establishing the store/load correlation dynamically at runtime, the proposed method establishes a small symbolic cache to capture existing syntax correlations and memory reference locality. The symbolic cache is addressed by the encoding content of store/load instructions to enable data accesses in the front end of the processor pipeline to shorten load-to-use latency. Performance evaluation of SPEC integer programs has demonstrated that the proposed method can achieve an accuracy over 70% with a small 4-kB symbolic cache. With compiler helps to reduce base register updates and to better utilize displacement values, further improvement of the SC accuracy may still be possible.

Acknowledgment

The authors would like to thank the anonymous referees for providing very helpful comments.

References

[1] T. Austin and G. Sohi, "Zero-cycle loads: microarchitecture support for reducing load latency," in *Proc. 28th Int. Symp. Microarchitecture*, Ann Arbor, MI, Dec. 1995, pp. 82–92.

[2] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz, and U. Weiser, "Correlated load-address predictors," in *Proc. 26th Int. Symp. Comput. 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,” in *Proc. 27th Int. Symp. Comput. Architecture*, Vancouver, Canada, June 2000, pp. 306–315.

[4] D. Burger and T. Austin, “The SimpleScalar Tool Set, Version 2.0,” Comput. Sci. Dept., Univ. Wisconsin-Madison, Tech. Rep. #1342, June 1997.

[5] B. Calder, G. Reinman, and D. Tullsen, “Selective value prediction,” in *Proc. 26th Int. Symp. Computer Architecture*, Atlanta, GA, May 1999, pp. 64–75.

[6] C. Chen and A. Wu, “Microarchitecture support for improving the performance of load target prediction,” in *Proc. 30th Int. Symp. Microarchitecture*, Triangle Park, NC, Dec. 1997, pp. 228–234.

[7] B. Cheng, D. Connors, and W. Hwu, “Compiler-directed early load-address generation,” in *Proc. 31st Int. Symp. Microarchitecture*, Dallas, TX, Dec. 1998, pp. 138–147.

[8] B. Chung, J. Zhang, J.-K. Peir, S. Lai, and K. Lai, “Direct load: dependence-linked dataflow resolution of load address and cache coordinate,” in *Proc. 34th Int. Symp. Microarchitecture*, Austin, TX, Dec. 2001, pp. 76–87.

[9] R. Eickemeyer and S. Vassiliadis, “A load-instruction unit for pipelined processors,” *IBM J. Res. Develop.*, vol. 37, no. 4, pp. 547–564, July 1993.

[10] J. Gonzalez and A. Gonzalez, “Speculative execution via address prediction and data prefetching,” in *Proc. 1997 Int. Conf. Supercomputing*, Vienna, Austria, Aug. 1997, pp. 196–203.

[11] ——, “The potential of data value speculation to boost ILP,” in *Proc. 1998 Int. Conf. Supercomputing*, Melbourne, Australia, June 1998, pp. 21–28.

[12] T. Horel and G. Lauterbach, “UltraSPARC-III: designing third-generation 64-Bit performance,” *IEEE Micro*, pp. 73–85, May/June 1999.

[13] K. Hua, A. Hunt, L. Liu, J.-K. Peir, D. Pruett, and J. Temple, “Early resolution of address translation in Cache design,” in *Proc. 1990 Int. Conf. Comput. Design*, Boston, MA, Sept. 1990, pp. 408–412.

[14] R. Kessler, “The Alpha 21 264 microprocessor,” *IEEE Micro*, vol. 19, no. 2, pp. 24–36, Mar./Apr. 1999.

[15] M. Lipasti, C. Wilkerson, and J. Shen, “Value locality and load value prediction,” in *Proc. 7th Int. Conf. Architectural Support Programming Languages Operating Syst.*, Boston, MA, Oct. 1996, pp. 138–147.

[16] M. Lipasti and J. Shen, “Exceeding the limit via value prediction,” in *Proc. 29th Int. Symp. Microarchitecture*, Paris, France, Dec. 1996, pp. 226–237.

[17] W. Lynch, G. Lauterbach, and J. Chamdani, “Low load latency through Sum-Addressed Memory (SAM),” in *Proc. 25th Int. Symp. Comput. Architecture*, Barcelona, Spain, June 1998, pp. 369–379.

[18] Q. Ma, J.-K. Peir, L. Peng, and K. Lai, “Symbolic Cache: fast memory access based on program syntax correlation of loads and stores,” in *Proc. 2001 Int. Conf. Comput. Design*, Austin, TX, Sept. 2001, pp. 54–61.

[19] A. Moshovos and G. Sohi, “Streamlining inter-operation memory communication via data dependence prediction,” in *Proc. 30th Int. Symp. Microarchitecture*, Triangle Park, NC, Dec. 1997, pp. 235–245.

[20] ——, “Read-after-read memory dependence prediction,” in *Proc. 32nd Int. Symp. Microarchitecture*, Haifa, Israel, Nov. 1999, pp. 177–185.

[21] D. Papworth, “Tuning the Pentium Pro Microarchitecture,” *IEEE Micro*, vol. 16, pp. 8–15, Apr. 1996.

[22] Y. Sazeides and J. Smith, “The predictability of data values,” in *Proc. 30th Int. Symp. Microarchitecture*, Triangle Park, NC, Dec. 1997, pp. 248–258.

[23] T. Slegel *et al.*, “IBM’s S/390 G5 microprocessor design,” *IEEE Micro*, vol. 19, no. 2, pp. 12–23, Mar./Apr. 1999.

[24] P. Song, “IBM’s Power3 to replace P2SC,” *Microprocessor Rep.*, vol. 11, no. 15, pp. 1–11, Nov. 1997.

[25] G. Tyson and T. Austin, “Improving the accuracy and performance of memory communication through renaming,” in *Proc. 30th Int. Symp. Microarchitecture*, Triangle Park, NC, Dec. 1997, pp. 218–227.

[26] K. Wang and M. Franklin, “Highly accurate data value prediction using hybrid predictors,” in *Proc. 30th Int. Symp. Microarchitecture*, Triangle Park, NC, Dec. 1997, pp. 281–290.

Lu Peng received the B.E. and M.S. degrees in computer science from Shanghai Jiaotong University, China. He is currently working toward the Ph.D. degree at the University of Florida, Gainsville.

His research interests include computer architecture, distributed systems, and computer networks.

Mr. Peng is a Member of the ACM.

Jih-Kwon Peir received the B.S. degree in engineering from Cheng-Kung University, Taiwan, the M.S. degree from the University of Wisconsin-Milwaukee, and the Ph.D. degree from the University of Illinois, Urbana-Champaign.

From 1986 to 1992, he was a Research Staff Member with IBM T. J. Watson Research Center, Yorkstown Heights, NY, involved in mainframe processor designs. From 1992 to 1993, he was the Deputy Director of Computer Technology, Industrial Technology Research Institute, Taiwan, in charge of an Intel Pentium-based SMP development project. Since 1995, he has spent several summers visiting Intel’s Microprocessor Research Lab and IBM’s Almaden Research Center. He is currently an Associate Professor in the Computer and Information Science and Engineering Department, University of Florida, Gainsville. His research interests include computer system architectures, designs, and performance evaluation.

Dr. Peir received the IBM Faculty Development Partnership Award in 1995 and an National Science Foundation Faculty Early Career Development Award in 1996. He was coauthor of two best paper awards in 1990 and 2001, in IEEE-ICCD conference. He serves as Editor of the *Journal of Parallel and Distributed Computing* and is on the editorial board of the IEEE TRANSACTIONS OF PARALLEL AND DISTRIBUTED SYSTEMS.

Qianrong Ma received the B.E. and M.E. degrees in electrical engineering from Tsinghua University, China, in 1993 and 1996, respectively. He received the M.S. degree in Computer Engineering, University of Florida, Gainsville, in May 2000.

He spent one year with Stone Corporation, China, as a Product Development Engineer. He is currently a Senior Member of the Technical Staff at Server Technology Division, Oracle Corporation, Redwood City, CA.

Konrad Lai received the B.S. degree in electrical engineering from Princeton University, Princeton, NJ, in 1976 and the M.S. degree in computer engineering from Carnegie Mellon University, Pittsburgh, PA, in 1978.

He is currently Principal Researcher and Manager in Microprocessor Research, Intel Labs, Hillsboro, OR. He has been with Intel for over 20 years, working on microprocessor, memory, and system architecture. He holds 25 patents and has authored or coauthored eight papers.

Mr. Konrad is a Member of ACM and the IEEE Computer Society.