Examination of a memory access classification scheme for pointer-intensive and numeric programs

(*mehrotra@csrd.uiuc.edu*)*
CSRD and UI Department of Computer Science
1308 West Main Street
Urbana, IL 61801–2307

Luddy Harrison
(*harrison@csrd.uiuc.edu*)
Connected Components Corporation
One Kendall Square, Building 200
Cambridge, MA 02139

Abstract

In recent work, we have described a data prefetch mechanism for pointer-intensive and numeric computations, and presented detailed measurements on a suite of benchmarks to quantify its performance potential[1] [HM94, Meh96]. In this paper we review a simple classification for memory access patterns on which the prefetch mechanism is based, and then take a close look at two codes from our suite. Focusing on just two programs allows us to display a wide range of simulation data. Results from this study several additional optimizations for future data prefetch mechanisms.

Keywords: CPU architecture, data cache, memory access pattern classification, instruction profiling, memory latency tolerance

1 Introduction

The ever-increasing gap between microprocessor and memory speeds has been well documented [HP96]. Instruction and data caches have become the principal means of bridging this speed discrepancy. Typically, first-level caches are small (8K to 32K bytes in size), direct mapped or modestly associative, and integrated on CPU chips. These design choices are made because the on-chip cache array access has to be completed within a single CPU clock cycle; the TLB lookup usually takes another cycle. Since secondary caches and main memory have traditionally been implemented separately from the CPU, and with SRAMs and DRAMs that can be much slower than the first-level cache, the ratio of first-level cache miss to hit times is growing. Loads and stores make up a large proportion of the instructions executed by typical programs. The interposition of a cache hierarchy between the CPU and main memory implies that memory accesses experience variable data latency, depending upon where in the memory hierarchy the desired data is found.

Additional increases in processor performance could be achieved if we could predict, a priori, the data reference patterns of loads involved in complex memory traversals, and then use this information to prefetch into the primary data cache, data for those loads that are responsible for the majority of misses. Data prefetching is a promising technique for tolerating the cache-miss latency in high performance processors. Hardware, software, and hybrid hardware-software schemes have all been extensively explored, both in the context of uniprocessors and multiprocessors[TS95, CB95, DS95, Gor95, CR94, Mow94, PK94, YGHH94, EV93, JT93, FPJ92, SH92, Sel92, KL91, GGV90, Jou90, Smi78].

For some programs, particularly scientific codes operating on dense arrays or matrices, data reference pattern prediction is easy. Consequently, several hardware prefetch mechanisms have been proposed for such codes, and effective compiler techniques developed for them [LRW91, Mow94, BCJ+94, CMT94]. Predicting the memory access patterns in pointer-intensive and sparse numerical computations is a much harder problem, and has received far less attention in the literature [TJ92, Sel92]. This is a significant omission, since both these types of codes generate memory access patterns that lead to poor data cache behavior. This is because compiler transformations to improve CPU memory hierarchy performance are typically based upon dependence testing of linear array index expressions in Fortran loop nests.

In this paper we first review a simple classification for memory access patterns on which the data prefetch mechanism is based, and then take a close look at two codes from our suite. Focusing on just two programs allows us to display a wide range of simulation data. Results from this study suggest additional optimizations for future data prefetch mechanisms.

The rest of this paper is organized as follows. In section 2 we discuss related work. In section 3 we detail our model for memory reference patterns generated by individual load instructions in programs. Section 4 provides a brief overview of the prefetch mechanism. Section 5 discusses our experimental methodology, and presents results for programs `Link-Gram` and `spice2g6`. Section 6 offers some conclusions from this research.

2 Related work

Related work is drawn from several topics of research. Closely related is the work by Abraham and Rau [AR94]. They reported results from the profiling of load instructions in the Spec89 benchmarks. They were interested in using the data to construct more effective instruction scheduling algorithms, and to improve compile-time cache management. Selvidge had similar goals in the experiments he reported in his thesis [Sel92]. Austin et al [APS95] profiled load instructions while developing software support for their fast address calculation mechanism. They reported aggregate data from their experiments, not individual instruction profiles. Lebeck and Wood used their CProf cache profiling system to analyze cache bottlenecks on a subset of the Spec92 codes [LW94]. They used the results to manually tune the codes using data structure and loop

*Currently with Sun Microsystems, Inc , Mountain View CA

[1]Aspects of this work are covered by a patent application filed by the UI Visit http://www.oc.uiuc.edu/rtmo for additional details

ICS'96, Philadelphia, PA,USA
© 1996 ACM 0-89791-803-7/96/05..$3.50

```
int i, m, a[100];
for (i=0; i<100; i++) { /* A */
    m = m + a[i];
}
```

Figure 1: Linear array traversal

```
int    n;
struct b { int x; double z; struct b *y; };
struct b *p, *q;
/* Construct list with SIZE elts */
q = build_list (SIZE);
/* Traverse it */
for (p=q; p!=NULL; p=p->y) {   /* B */
   n = n + p->x;
}
```

Figure 2: Linked-list traversal

```
int i, x;
int c[N], d[N];
/* c is sparse, d is c's index array */
i = index_of_head_of_list;
while (i)   {   /* C */
   x = x + c[i];
   i =  d[i];     /* update pointer */
}
```

Figure 3: Sparse linked-list traversal

```
int i, x;
int c[N], d[N];
/* c is sparse, d is c's index array */
for (i=0; i < 10; i++) {   /* D */
   x = x + c[d[i]];
}
```

Figure 4: An indirection-vector based sparse representation

transformations for direct mapped caches. As mentioned earlier, many data prefetching schemes have been proposed in the literature, using hardware, software, or hybrid techniques. Some of these have provided classifications of load instructions, but almost always focused on scientific codes. None of these studies has proposed a model to explain load behavior across a broad range of programs, as our work does.

3 Classifying load instructions

This section describes our load classification model. We will illustrate it using code fragments written in C. First, consider the code in Figure 1 that performs a reduction on array a. In loop A, every element of the array a is added to m. When executing, loop A will generate the memory addresses (ignoring the scalars i and m, and instruction addresses)

```
a, a+4, a+8, a+12, a+16, a+20, a+24, ...
```

and so on. This sequence of addresses can be described by the first order linear recurrence

$$a_k = a_{k-1} + 4,\ k \in \{1,2,3,\ldots\} \qquad (1)$$

We call this a *linear address sequence*. Loops of this type are common in dense numeric programs.

Next, consider a reduction on elements of a singly-linked list, illustrated in Figure 2. Symbolic programs are distinguished by their extensive use of pointer-linked data structures. This loop, when executed, will generate the memory addresses (ignoring the scalar n, and instruction addresses)

```
*q, *q+12, *(*q+12), *(*q+12)+12,
*(*(*q+12)+12), *(*(*q+12)+12)+12, ...
```

because every x field in the linked list pointed to by q is added to n. Note that in the above sequence, *(*q+12) represents a single address, given in terms of the initial value of the variable q, and not an expression evaluation that involves two memory references and an addition. When we consider the addresses in the above sequence that correspond to updates of pointer p (every other address starting with the second one)

```
*q+12, *(*q+12)+12, *(*(*q+12)+12)+12, ...
```

we see that it too can be described by a first order recurrence, given by

$$p_k = \text{Mem}[p_{k-1}] + 12,\ k \in \{1,2,3,\ldots\} \qquad (2)$$

We call this an *indirect address sequence*. Here, $\text{Mem}[p_{k-1}]$ refers to the contents of the memory location pointed to by p, i.e. *p. The index variable k is used to denote successive values of p.

Consider a reduction once again, this time on a sparse vector, c, and its associated index array, d. If the representation used is one that simulates linked lists using arrays, the code might resemble the fragment shown in Figure 3. When executing, loop C will issue the memory addresses (ignoring the scalars i and x, and instruction addresses)

```
c+4i, d+4i, c+4(*(d+4i)), d+4(*(d+4i)), ...
```

and so on. This loop is representative of code found in some sparse numeric programs. As in the linked-list example, note that c+4(*(d+4i)) represents a single address, given in terms of the starting address of the array c and array elements d[i]. The addresses for accessing elements of d also describe a first order recurrence

```
d+4i, d+4(*(d+4i)), d+4(*(d+4(*(d+4i)))), ...
```

This recurrence can be expressed by the equation

$$d_k = 4 \times \text{Mem}[d_{k-1}] + \text{Base(d)},\ k \in \{2,3,4\ldots\} \qquad (3)$$

where Base(d) is the base address in memory of index array d. d_1 is set before loop C is entered. Notice that Equation (3) represents an indirect address sequence similar to the recurrence for the pointer-chasing example (Equation (2)), the difference being the component that varies. Here, the base address of array d is fixed, and we are accessing elements of d randomly. In the linked-list traversal, the base address of each object retrieved from memory varies (as we step through the heap randomly); however, the offset within each object where the pointer to the next object is to be found is fixed.

Numerous other sparse representations exist [DER86]. Some use linked structures to index the sparse array, as in Figure 3, while others use indirection vectors for storing the indices of nonzero elements. An example of the latter representation is shown in Figure 4. In this case, accesses to array d describe a linear address sequence as described by Equation (1).

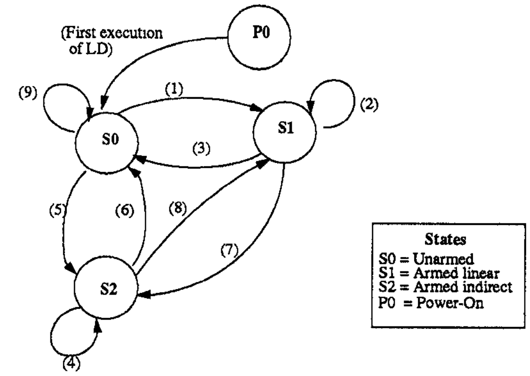
Clearly many load instructions in a program image will not obey Equations (1), (2), and (3). A partial list of such loads includes those involved in scalar accesses, loads that access non-pointer data fields of structures, and register reloads at subroutine returns. However, what makes the classification valuable in spite of this limitation, is the fact that *prefetching cache lines containing well predicted loads is often sufficient to mask a significant number of cache misses due to loads that are not predicted by our model.* This effect is due to the spatial locality afforded by the prefetched cache lines.

4 The Indirect reference buffer

The *indirect reference buffer* (IRB) is a device that exploits recurrent patterns of memory access (like those exhibited by loops A through E of section 3) for prefetching. In this section we briefly describe the IRB; see [Meh96] for more details. The IRB is organized as two mutually cooperating sub-units: a *recurrence recognition unit* (RRU) and a *prefetch unit* (PU). The RRU recognizes linear address sequences and indirect address sequences such as those described by Equations (1), (2), and (2), and having recognized them, directs the PU to load data into the primary data cache in anticipation of addresses the processor will issue. The RRU consists of a table, the Reference Prediction Table (RPT), a couple of adders and comparators, logic to implement a finite state machine, and a set of buffers to store intermediate data for load instructions being concurrently processed by the CPU pipeline. Similarly, the PU consists of a table, the Active Prefetch Buffer (APB), and a collection of simple logic circuits. For the purposes of this paper, however, it is sufficient to consider a *logical* IRB comprised of a reference prediction table and a state machine.

The entries in the reference prediction table are indexed by the virtual addresses of load instructions. Each entry consists of several fields, the first of which is the instruction address. The second field is the (virtual) operand address last issued by this load instruction. The third field is the register contents returned from memory for this load instruction the last time it executed. The fourth field contains a *linear address stride* computed by subtracting the previous addresses issued for this instruction from the current one. The fifth field contains an *indirect address stride* computed by subtracting the previous contents returned for this load from the current address. All address stride calculations are performed using unsigned integer arithmetic. The sixth and final field contains state information that is used to arm the RRU, as well as the load opcode.

Figure 5 shows the transition diagram for the IRB state machine. This state machine is designed such that for any particular load, it will generate prefetches using either the indirect address stride or the linear address stride at any one time, not both. It is basically a combination of two simpler state machines, one of which checks the stability of the linear stride, while the other concurrently checks the stability of the indirect stride. This arrangement allows each load to be checked simultaneously for a linear or indirect address pattern.



Transitions

(Compute both linear and indirect strides in each state)

State S0

(1) If new linear stride == previous, and linear stride != 0, go to S1.

(5) If not going to S1, and new indirect stride == previous, and load contents != 0, go to S2.

(9) If no transition to S1 or S2, stay in S0.

State S1

(2) If new linear stride == previous, and linear stride != 0, stay in S1. *Generate prefetch.*

(7) If not staying in S1, and new indirect stride == previous, and load contents != 0, go to S2.

(3) If no transition to S1 or S2, return to S0. *Disable prefetching.*

State S2

(8) If new linear stride == previous, and linear stride != 0, go to S1.

(4) If not going to S1, and new indirect stride == previous, and load contents != 0, stay in S2. *Generate prefetch.*

(6) If no transition to S1 or S2, return to S0. *Disable prefetching.*

Figure 5: State transition diagram for the IRB state machine

5 Experimental evaluation

In this section we describe our experimental framework, and present some simulation results.

5.1 Simulation methodology

Both programs are compiled with standard optimization, and the resulting executables instrumented using Qpt [Lar93]. We have modified Qpt so that in addition to generating instruction and data traces, it also generates the contents of all memory locations that are read, a unique identifier (an integer) for each load when it executes, and the load opcode type (byte, half, or word load). Using Qpt allows us to trace all user mode program references (including library routines) but no operating system code. All experiments reported have been performed on MIPS R3000/R3010 based DEC workstations running Ultrix 4.2A, and using the GNU C compiler (gcc) version 2.5.8.

To gather dynamic load profiles, we maintain a reference prediction table that records data for all possible loads in the program image. When a load instruction is traced, its (unique) identifier is used to locate its entry in the table and update the fields. At this time, a future operand address is also predicted for the load if it is in the midst of a linear or indirect address sequence, using the state machine and equations similar to (1), (2), or (2). In addition, several statistics gathering fields are associated with each load entry. We use these fields to record quantities such as the load execution count, the number of cache misses each load causes in a particular cache configuration, the number of times it was involved in a linear or indirect memory sequence, a histogram of the load's (absolute) linear address strides, and so on.

To calibrate the cache behavior of the program in terms of individual load instructions, and to determine if this behavior was sensitive to cache organization, we examined a broad range of realistic first-level data caches. For all experiments, the cache line size was fixed at 32 bytes, and the replacement policy chosen as LRU. Thereafter, cache size was varied as 8K or 32K, set associativity chosen from one, two, four, eight, or full (256-way for 8K, 1024-way for 32K), and the cache replacement and memory update policy varied as write through with no write allocate, or write back with write allocate. This resulted in sixteen load profiles[2].

[2] In fact, many more profiles were actually constructed, as state machines for detecting load reference patterns were perfected, and a variety of other tradeoffs examined. These experiments are beyond the scope of this paper, and are reported in the first author's dissertation [Meh96].

Code and Input Set	Link-Gram examples.batch — 397 English sentences	spice2g6 greycode.in — short transient analysis
Load classification statistics for all experiments		
# static	4467	25897
# activated	3413	5296
# executed thrice	3294	3998
# linear	1141	795
# indirect	58	8
# both	406	76
Dynamic instruction counts for all experiments		
# Instructions	629,335,554	3,562,033,982
# Reads (LDs)	122,825,560	774,857,452
# Writes (STs)	62,151,589	151,471,820

Simulation data for various cache configurations

Configuration (Id)	Link-Gram # Misses		Link-Gram #LDs causing			spice2g6 # Misses		spice2g6 #LDs causing		
	Read	Write	25%	50%	75%	Read	Write	25%	50%	75%
8K/direct/wt/nwa (1)	11,510,076	5,674,869	6	29	88	198,824,439	39,121,368	2	3	13
8K/direct/wb/wa (2)	10,703,942	1,714,180	5	25	80	196,021,859	7,059,673	2	3	11
8K/2-way/wt/nwa (3)	8,720,970	4,718,132	4	18	57	176,232,802	31,793,355	2	3	10
8K/2-way/wb/wa (4)	8,123,188	1,235,518	4	16	49	173,877,368	5,261,481	2	3	9
8K/4-way/wt/nwa (5)	7,934,329	4,454,465	3	15	49	163,367,296	29,030,273	2	3	8
8K/4-way/wb/wa (6)	7,399,948	1,127,910	3	13	43	161,728,333	4,459,421	2	3	7
8K/full/wt/nwa (7)	7,324,072	4,343,573	3	13	43	159,473,886	28,618,910	2	3	7
8K/full/wb/wa (8)	6,814,840	1,082,373	3	11	37	157,991,665	4,314,389	2	3	7
32K/direct/wt/nwa (9)	5,773,555	3,757,965	4	17	63	121,028,013	24,603,501	2	3	9
32K/direct/wb/wa (10)	5,242,905	994,085	3	15	54	119,492,973	3,616,322	2	3	8
32K/2-way/wt/nwa (11)	**3,994,660**	**3,057,682**	**2**	**11**	**35**	**71,390,985**	**23,027,729**	**2**	**4**	**17**
32K/2-way/wb/wa (12)	3,610,008	685,993	2	9	29	70,171,228	3,262,295	2	4	16
32K/4-way/wt/nwa (13)	3,593,987	2,951,664	2	9	28	46,639,971	22,409,669	4	9	29
32K/4-way/wb/wa (14)	3,222,013	652,627	2	8	23	45,656,602	2,883,110	4	9	27
32K/full/wt/nwa (15)	3,319,106	2,823,737	2	9	24	39,755,429	22,279,147	5	12	33
32K/full/wb/wa (16)	3,319,106	2,823,737	2	9	24	38,848,592	2,813,350	5	11	31

Table 1: **Aggregate statistics for `Link-Gram` and `spice2g6` experiments, measured on DECstation 5000s running Ultrix 4.2A.** wt = write through, nwa = no write allocate, wb = write back, wa = write allocate. The *#LDs causing* columns show the number of loads that are responsible for 25%, 50%, and 75% of the read misses. These quartile load distributions are with respect to the read misses in the same row. Configuration 11 (in bold) is used for the detailed load profile data in Tables 2 and 3.

5.2 Results and analysis

5.2.1 Aggregate data

Table 1 lists some aggregate characteristics and miss rates for all cache configurations of `Link-Gram` and `spice2g6` that we studied. To give the reader some idea about the cache miss distribution over loads, quartile distributions for loads are also given.

The data in Table 1 is divided into four sections. The first section lists the program and input data used. The second section classifies the load instructions in the program image. The row labeled *# static loads* is the total number of loads in the executable detected during instrumentation. The row labeled *# loads activated* is the number of loads instructions that were executed at least once while processing the given input data set. Loads executed at least three times are listed in the row labeled *# executed thrice*. Three is the minimum number of times a load has to execute for any address patterns to be detected[3]. The rows with labels *linear*, *indirect* and *both* show the number of loads that were involved in linear address sequences, indirect address sequences, or both. Surprisingly, it is not uncommon to find loads that participate in both types of memory address sequences at different times during a program's execution[4]. Finally, the difference between the *# executed thrice* and the sum of the *# linear*, *# indirect*, and *# both* columns is the number of loads that were not involved in either kind of memory traversal. The third section of Table 1 provides the dynamic reference counts for instructions, memory reads, and memory writes, that are common to all our simulations. The fourth section shows the read and write misses, and the quartile distribution of the contribution of individual loads to the overall read miss count, for each cache configuration simulated.

Several interesting facts can be noted from the load characteristics section of Table 1. Only a small fraction, typically between one-tenth and one-third, of the total number of static loads in a program get activated for a typical input set. There is a further drop when we isolate those that execute at least three times, and a dramatic further drop when we look at those that follow any of the patterns recognized by the IRB. While both codes contain loads that are linear, the symbolic codes also have a significant number of indirect loads, as expected. The rather large number of loads that are not classified in any category for `spice2g6` prompted us to examine the number of read misses they contributed. The number was negligible, with all 3119 non-classified loads together contributing less than 1.5% of the total read misses.

5.2.2 Load classification data

For presenting the load profile data, we chose as our reference configuration, a 32K byte, 2-way associative data cache with LRU replacement, 32 byte lines and no subblocks, and a write through with no write allocate write policy. In our opinion, this is a reasonable limiting size for what we expect a practical first level CPU

[3] Note that a load that executes once or twice denotes a trivial linear sequence.

[4] A common example is when a dynamic data structure is constructed using multiple calls to `malloc()`. Since many memory allocators first try to allocate memory from lists of blocks of fixed sizes, a linked data structure can often appear to be linear because its records are a constant distance apart in the program's address space.

data cache to be in the next few years, and our simulation results have shown that the use of even modest associativity is sufficient for dumping prefetched data directly into such a cache. This design decision allows us to allows us to avoid the complexities introduced by conflict misses [AR94, LW94].

We now examine the twenty most heavily missed loads in `Link-Gram` and `spice2g6`. This data is presented in Tables 2 and 3 respectively. Each entry in these two tables has eleven fields. The first field, labeled *Load Id #*, is the unique identifier assigned to each load during instrumentation of the executable. The second field is the name of the routine in which the load occurs. The third, labeled *Op. Type*, is a mnemonic representing the type of load. Since we instrumented programs running on MIPS R3000/R3010 based DEC workstations, the possible load types are `LB, LBU, LH, LHU, LW, LWL, LWR` for integer values, and `LWC1` for floating point quantities. On this CPU double precision operands are loaded using two consecutive `LWC1` instructions. The fourth field, labeled *How Armed*, shows the different memory address sequences in which this load was involved. The possible mnemonics for this field are `LIN, IND, BOTH, BOTH+LI` and `NONE`. If this field has the mnemonic `LIN`, it means the load participated only in linear address sequences for this input set. Likewise, a mnemonic of `IND` says that the load was only involved in indirect address sequences. `BOTH` means that the load participated in both types of sequences, while `BOTH+LI` means that it was involved in sequences that were simultaneously linear and indirect. Finally, `NONE` means that the load was not involved in any recognizable address sequence.

The fifth field in Tables 2 and 3, *Execution Count*, lists the total execution count for each load[5]. The sixth and seventh fields (*Linear Count* and *Indirect Count*, respectively) list the number of times each load was involved in linear or indirect address sequences respectively. Byte and half word loads are prohibited from participating in indirect address sequences, since their contents cannot be used to compose meaningful pointer addresses. The eighth field, labeled *Zero Stride*, counts the number of times successive operand addresses generated by a load were identical. A high count in this field (relative to the total load execution count in column five) implies that a scalar variable is being accessed. The ninth field, labeled *# Rd Misses Pre*, lists the read misses generated by each load for a particular cache configuration, in this case, configuration (11). The count in the tenth field, labeled *Succ. Predictions*, is an indicator of how strongly a load is following one of the recognized memory access patterns. If the load is involved in a linear or indirect memory address sequence, the appropriate future operand address is predicted, and at the next execution of the load, this address is compared with the actual operand address. If there is a match, the success count is incremented. The final field, *# Read Misses Post*, shows the number of misses generated by the loads after the cache has been primed with lines containing the addresses predicted by the IRB state machine. Only one cache line is prefetched for each prefetch request, and it is marked MRU when placed in the cache. The $+$ or $-$ sign in parentheses following the read miss count is used to indicate how the load's behavior was affected by prefetching. A $+$ indicates that the load experienced more misses after prefetching was enabled, while a $-$ indicates a reduction.

From the *# Read Misses Pre* columns of Tables 2 and 3, we find that approximately ten loads account for 50% of the total read misses for these two codes. Also, from Table 1 it can be observed that if we exclude the direct mapped 8K caches[6], then less than twenty loads are responsible for over 50% of the misses, regardless of the cache configuration chosen. As expected, the number of read misses experience is lower for the 32K caches, since they

[5]The total dynamic count for each code is listed in Table 1.

[6]In which many loads are clearly experiencing conflict and capacity misses, thereby spreading the read misses over a larger number of loads

```
813:int
814:dict_match (char *s, char *t)
815:{
826:  while ((*s != '\0') && (*s == *t)) {
828:      s++;
829:      t++;
830:  }
    lb r3,0(r4)    <== #827
    nop
    addiu r5,r5,1
    beq r3,r0,0x405fe0
    nop
    lb r2,0(r5)    <== #828
    nop
    beq r3,r2,0x405fb8
    addiu r4,r4,1
    addiu r4,r4,-1
831:  if ((*s == '*') || (*t == '*'))
832:      return 0;
833:  return (((*s == '.') ? ('\0') :
      (*s)) ((*t == '.') ? ('\0') : (*t)));
      ..........
     }
```

Figure 6: MIPS R3000 assembly code for segment containing the heavily-missed load #828 in routine `dict_match()` from `Link-Gram`

capture more of the cache working set of the program for the given input data set. Looking down the successful predictions columns of Tables 2 and 3, we find considerable variability in the prediction accuracy. This is to be expected; our model does not predict all loads well in pointer-intensive and sparse numeric codes. It will, and does, predict all the heavily missed loads very accurately in dense numeric codes.

The key in the case of symbolic and sparse codes, is that the well predicted loads are used to cover many of the misses generated by the poorly predicted loads. Some evidence of this is provided by the *#Read Misses Post* columns of Tables 2 and 3. Upon comparing this column with the *# Read Misses Pre* column of Table 2 for `Link-Gram` on a load-by-load basis, we find that several of the loads have experienced substantial reductions in their number of misses, and only one of these loads, #4091, was well predicted. Although over half of the loads show increases in the their number of misses, most increases are actually negligible (less than 1%). Repeating this exercise with the corresponding columns of Table 3 for `spice2g6`, we find that almost all the loads show a reduction in their number of misses, in spite of the variability in prediction accuracy. However, unlike `Link-Gram`, no load shows a dramatic reduction in miss count for `spice2g6`. There is additional experimental evidence to show that this *miss covering* property of the well predicted loads can be consistently exploited during prefetching, for several important codes [Meh96].

5.2.3 Analysis of code fragments

To get a good understanding of the nature of the load misses in `Link-Gram` and `spice2g6`, and to understand why load prediction accuracy is highly variable, we will examine source code and dis-assembled MIPS assembly code for routines containing several of the loads from Tables 2 and 3. While examining these code fragments, it should be remembered that the MIPS R3000 processor used in our DECstations has a branch delay slot of one cycle, and a load delay slot of one cycle. The GNU C compiler attempts to fill both delay slots whenever possible. If it fails, it generates a `nop`.

Load Id #	Routine Name	Op. Type	How Armed	Execution Count	Linear Count	Indirect Count	Zero Stride	# Rd Misses Pre	Succ. Pred.	# Rd Misses Post
4091	`malloc()`	LW	BOTH+LI	1471087	44315	1196763	4386	676565	1121199	92417(−)
4098	`free()`	LBU	LIN	1371691	3087	0	19354	365503	658	366824(+)
828	`d_m()`	LB	LIN	1657228	597735	0	1	149013	185812	149366(+)
1466	`m_d_c()`	LB	LIN	445277	141	0	0	140649	18	141810(+)
1470	`m_d_c()`	LW	BOTH+LI	420666	122	592	14	131510	6	132764(+)
1763	`c_E()`	LW	LIN	218794	61895	0	0	116169	18247	96453(−)
1753	`s_o_e()`	LB	LIN	201893	63	0	0	105040	9	105255(+)
1755	`s_o_e()`	LW	BOTH+LI	191160	53	293	0	99753	2	99997(+)
863	`r_l()`	LW	BOTH+LI	401998	5644	10536	2	94194	187	81338(−)
1604	`c_t()`	LW	BOTH+LI	159163	94	26	0	87034	1	36175(−)
1119	`reverse()`	LW	BOTH+LI	102814	692	1990	2	83278	267	82373(+)
1731	`f_c()`	LW	BOTH+LI	195193	826	4608	42	80326	1003	80622(+)
1745	`f_E()`	LB	LIN	166066	604	0	5	71791	147	72224(+)
855	`r_d_l()`	LW	BOTH+LI	180512	3787	4099	795	69559	0	58324(−)
1749	`f_El()`	LW	LIN	159489	72	0	4	69470	0	69881(+)
1606	`c_t()`	LH	LIN	159163	111	0	1	55352	0	55661(+)
1770	`c_E_l()`	LW	NONE	207774	0	0	0	52860	0	40433(−)
1572	`s_d_f()`	LW	BOTH+LI	110190	308	2089	4	47867	151	47936(+)
1278	`hash_S()`	LH	LIN	199926	10672	0	33493	39967	531	39927(+)
1280	`hash_S()`	LBU	LIN	199926	31571	0	11735	38077	8359	35883(−)

Table 2: **Detailed profiles for the twenty most heavily missed loads in `Link-Gram` for reference cache configuration** — a 32K byte, 2-way associative data cache with LRU replacement, 32 byte lines and no subblocks, and a write through with no write allocate write policy. `c_E()`: `copy_Exp()`, `c_t()`: `clean_table()`, `d_m()`: `dict_match()`, `f_c()`: `free_connectors()`, `f_E()`: `free_Exp()`, `f_El()`: `free_Elist()`, `m_d_c()`: `mark_dead_connectors()`, `r_l()`: `rabridged_lookup()`, `s_d_f()`: `set_dist_fields()`, `s_o_e()`: `size_of_expression()`. See the text of Section 5.3.2 for a description of the columns, and Table 1 for aggregate characteristics of `Link-Gram`.

Load Id #	Routine Name	Op. Type	How Armed	Execution Count	Linear Count	Indirect Count	Zero Stride	# Rd Misses Pre	Succ. Pred.	# Rd Misses Post
3124	`dcdcmp()`	LW	BOTH	98101898	456039	13	385184	10586854	10644	10581180(−)
3125	`dcdcmp()`	LW	BOTH	98101898	392100	7996	385176	9787664	10644	9777508(−)
3121	`dcdcmp()`	LW	BOTH	39445586	88729	13309	397811	8838460	10649	8839242(+)
3120	`dcdcmp()`	LW	BOTH	39445586	74535	3549	397832	7432427	0	7435033(+)
6951	`indxx()`	LW	BOTH+LI	7943578	56175	4273	503	2618788	16927	2618503(−)
6950	`indxx()`	LW	BOTH	7943581	113683	26	493	2576455	53790	2566632(+)
3639	`dcsol()`	LWC1	LIN	2366520	83472	0	0	1686417	11544	1675704(−)
3131	`dcdcmp()`	LWC1	LIN	7013424	130623	0	137517	1443697	10656	1438467(−)
3645	`dcsol()`	LW	BOTH	2366520	83472	12432	0	1322286	11544	1317009(−)
3679	`dcsol()`	LWC1	LIN	2362080	95016	0	0	1254996	22200	1246222(−)
3675	`dcsol()`	LW	BOTH	3259848	15983	4440	0	1113579	2664	1112691(−)
3674	`dcsol()`	LW	LIN	3259848	59495	0	0	973659	26640	967231(−)
3646	`dcsol()`	LW	BOTH+LI	2366520	29304	1777	0	946267	12432	943550(−)
3129	`dcdcmp()`	LWC1	LIN	7013424	239695	0	0	873517	10656	866886(−)
3110	`dcdcmp()`	LW	BOTH+LI	7013424	239695	15101	0	816792	10656	806212(−)
3134	`dcdcmp()`	LW	LIN	7013424	239695	0	0	775383	10656	765533(−)
3102	`dcdcmp()`	LWC1	LIN	2366520	100276	0	0	648061	23974	631400(−)
6949	`indxx()`	LW	BOTH+LI	1948098	829496	6	269	581251	618134	507856(−)
2988	`dcdcmp()`	LWC1	LIN	1026437	24195	0	0	567114	1609	565913(−)
3654	`dcsol()`	LWC1	LIN	905760	91463	0	0	513675	15984	501243(−)

Table 3: **Detailed profiles for the twenty most heavily missed loads in `spice2g6` for reference cache configuration** — a 32K byte, 2-way associative data cache with LRU replacement, 32 byte lines and no subblocks, and a write through with no write allocate write policy. See the text of Section 5.3.2 for a description of the columns, and Table 1 for aggregate characteristics of `spice2g6`.

```
          ..........
204: C
205: C      LOCATE ELEMENT (I,J)
206: C
207:    135 IF (J.LT.I) GO TO 145
208:        LOCIJ=LOCC
209:    140 LOCIJ=NODPLC(IRPT+LOCIJ)
210:        IF (NODPLC(IROWNO+LOCIJ).EQ.I)
          1 GO TO 155
211:        GO TO 140
212:    145 LOCIJ=LOCR
213:    150 LOCIJ=NODPLC(JCPT+LOCIJ)
214:        IF (NODPLC(JCOLNO+LOCIJ).EQ.J)
          1 GO TO 155
215:        GO TO 150
216:    155 VALUE(LVN+LOCIJ)=VALUE(LVN+LOCIJ)-
217:      1    VALUE(LVN+LOCC)*VALUE(LVN+LOCR)
218:    160 LOCC=NODPLC(JCPT+LOCC)
219:        GO TO 130
220:    170 LOCR=NODPLC(IRPT+LOCR)
221:        IF (IPIV.LE.0) GO TO 125
222:        NODPLC(NUMOFF+I)=NODPLC(NUMOFF+I)-1
223:        GO TO 125
          ..........
```

Figure 7: Fortran source code for segment containing the heavily missed loads #3120, #3121, #3124, #3125, #3129, #3131, and #3134 in routine `dcdcmp()` from `spice2g6`

`Link-Gram` From Table 2 it can be observed that LD #4091 and LD #4098 come from the library routines `malloc()` and `free()` respectively, for which we do not have access to the source code. Examination of their assembly code would add little to this discussion. We simply note that LD #4091 is well predicted. On the contrary, LD #4098 is poorly predicted for the same reasons that limit prediction accuracy in the routines we discuss below. Continuing with routine `dict_match()`, which contains LD #828 (address 0x405fcc), the listing in Figure 6 shows that this is a byte load that is dereferencing a pointer passed in as a parameter to the routine. Register `r5` holds the pointer to string `t` when `dict_match` is called. Depending upon the caller of this routine, `r5` can have a value completely unrelated to its previous value, which makes it hard to predict operand addresses for this load. This is also the reason this load misses so heavily. -

`spice2g6` Consider the routine `dcdcmp()`, which contains LD #3120, #3121, #3124, #3125, #3129, #3131, and #3134 from amongst the top-twenty missed loads. The source code for the fragment from `dcdcmp()` that contains these loads is shown in Figure 7. The first four of these loads contribute over half (51%) of the read misses for our reference cache configuration. The function of routine `dcdcmp()` is to swap rows and columns in the Y-matrix in accordance with the numerical pivoting requirements, and then to perform an in-place LU factorization of the Y-matrix. As the comment with the fragment suggests, the illustrated code is used for locating elements from the Y-matrix. The four heavily missed loads are used in the array index calculations for array `NODPLC` on lines 209–210, and 213–214. A quick study of the code shows that elements of `NODPLC` are being accessed with no spatial locality, which explains their poor predictability.

5.3 Implications for data prefetch mechanism design

Based on the data analyzed in this paper, several observations can be made. First, we showed that for both programs a very small number of load instructions[7] contributed over half of all read misses, for a wide range of first-level cache configurations. This suggests that most of the gains from prefetching can be had by focusing our efforts on these heavily-missed loads. Second, we found that the proposed model classifies only a subset of the eligible loads in program executables. Therefore it is as important to throttle prefetch generation for poorly predicted loads, as it is to exploit the well predicted ones. Third, we observed that many loads in real-world programs, such as `Link-Gram` and `spice2g6`, vary dynamically , following both the linear address sequence and the indirect address sequence at different times. This implies that prefetch devices that can adapt to this variation will be far more effective than those that are hardwired to follow one or the other.

Finally, we noted that there is considerable variability in the prediction accuracy of heavily-missed load instructions in pointer-intensive and numeric programs. This seems perplexing at first, because computer architects are used to seeing results for branch predictors and dense numeric code data prefetch mechanisms, where prediction accuracy is very high (typically over 90%). However, as discussed earlier, only a small number of loads in symbolic and sparse programs will be well predicted by our model, because of the specific recurrences that are being sought. For a prefetch device based upon this model to be effective, it is sufficient to prefetch cache lines just for the well predicted loads.

6 Conclusions

In this paper, we took a close look at a classification of memory access patterns for load instructions. To build insight into our model, detailed simulation data was presented and analyzed for two non-trivial symbolic programs. Exemplary code fragments extracted from the source distribution of the programs were also examined to illustrate the model. Finally, the implications of this classification on the design of general purpose data prefetch mechanisms were briefly discussed.

Acknowledgements

This work made extensive use of the QPT program tracing and profiling tool developed by James Larus of the University of Wisconsin at Madison. The authors would like to thank Jim for assisting them in making the necessary modifications to QPT to support this research.

References

[APS95] Todd M. Austin, Dionisios N. Pnevmatikatos, and Gurindar S. Sohi. Streamlining Data Cache Access with Fast Address Calculation. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 369–380, June 1995.

[AR94] Santosh G. Abraham and B. Ramakrishna Rau. Predicting Load Latencies Using Cache Profiling. Technical Report HPL–94–110, Hewlett-Packard Laboratories, Palo Alto, CA, November 1994.

[BCJ+94] David F. Bacon, Jyh-Herng Chow, Dz-ching R. Ju, Kalyan Muthukumar, and Vivek Sarkar. A Compiler Framework for Restructuring Data Declarations to Enhance Cache and TLB Effectiveness. In *Proceedings of CASCON '94*, pages 270–282, Toronto, Canada, October 1994.

[7] Compared to the total number present in the program executable.

[CB95] Tien-Fu Chen and Jean-Loup Baer. Effective Hardware-Based Data Prefetching for High-Performance Processors. *IEEE Transactions on Computers*, 44(5):609–623, May 1995.

[CMT94] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler Optimizations for Improving Data Locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, October 1994.

[CR94] Mark J. Charney and Anthony P. Reeves. Correlation-Based Hardware Prefetching. Submitted to IEEE Transactions on Computers, September 1994.

[DER86] I. S. Duff, A.M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices.* Oxford University Press, New York, NY, 1986. Printed in paperback (with corrections) 1989.

[DS95] Fredrik Dahlgren and Per Stenström. Effectiveness of Hardware-Based Stride and Sequential Prefetching in Shared Memory Multiprocessors. In *Proceedings of the first IEEE Symposium on High-Performance Computer Architecture*, pages 68–77, January 1995.

[EV93] Richard J. Eickemeyer and S. Vassiliadis. A load-instruction unit for pipelined processors. *IBM Journal of Research and Development*, 37(4):547–564, July 1993.

[FPJ92] John W. C. Fu, Janak H. Patel, and Bob L. Janssens. Stride Directed Prefetching in Scalar Processors. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 102–110, December 1992.

[GGV90] Edward H. Gornish, Elana D. Granston, and Alexander V. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, pages 354–368, Department of Computer Science, Urbana, IL 61801, June 1990.

[Gor95] Edward H. Gornish. *Adaptive and integrated data cache prefetching for shared-memory multiprocessors.* PhD thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, Urbana, IL 61801, January 1995.

[HM94] Luddy Harrison and Sharad Mehrotra. A data prefetch mechanism for accelerating general-purpose computation. Technical Report 1351, CSRD, University of Illinois at Urbana-Champaign, Urbana, IL 61801, 8 May 1994. Last revised 9 March 1995. This report is the basis for Patent Application No. 08/508,290, *Prefetch System Applicable to Complex Memory Access Schemes*, filed by the University of Illinois on 27 July 1995.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Mateo, CA 94403, second edition, 1996.

[Jou90] Norman P. Jouppi. Improving Direct-mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 364–373, May 1990.

[JT93] Ivan Jegou and Olivier Temam. Speculative Prefetching. In *Proceedings of the 1993 ACM International Conference on Supercomputing*, pages 57 – 66, July 1993.

[KL91] Alexander C. Klaiber and Henry M. Levy. An architecture for software-controlled data prefetching. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 43–53, May 1991.

[Lar93] James R. Larus. Efficient Program Tracing. *IEEE Computer*, 26(5):52–61, May 1993.

[LRW91] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, April 1991.

[LW94] Alvin R. Lebeck and David A. Wood. Cache Profiling and the SPEC Benchmarks: A Case Study. *IEEE Computer*, 27(10):15–26, October 1994.

[Meh96] Sharad Mehrotra. *Data prefetch mechanisms for accelerating symbolic and numeric computation.* PhD thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, Urbana, IL 61801, May 1996.

[Mow94] Todd C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching.* PhD thesis, Stanford University, Department of Electrical Engineering, Stanford, CA 94305, March 1994.

[PK94] Subbarao Palacharla and Richard E. Kessler. Evaluating Stream Buffers as a Secondary Cache Replacement. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 24–33, April 1994.

[Sel92] Charles William Selvidge. *Compilation-Based Prefetching for Memory Latency Tolerance.* PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, MA 02139, May 1992.

[SH92] James E. Smith and Wei-Chung Hsu. Prefetching in Supercomputer Instruction Caches. In *Proceedings of Supercomputing '92*, pages 588–597, November 1992.

[Smi78] Alan Jay Smith. Sequential Program Prefetching in Memory Hierarchies. *IEEE Computer*, 11(12):7–21, December 1978.

[TJ92] Olivier Temam and William Jalby. Characterizing the Behavior of Sparse Algorithms on Caches. In *Proceedings of Supercomputing '92*, pages 578–587, November 1992.

[TS95] John Tse and Alan Jay Smith. Performance Evaluation of Cache Prefetch Implementation. Technical Report UCB/CSD–95–877, Computer Science Division, University of California, Berkeley, CA 94720, June 1995.

[YGHH94] Yoji Yamada, John Gyllenhall, Grant Haab, and Wen-mei W. Hwu. Data Relocation and Prefetching for Programs with Large Data Sets. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, November 1994.