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

Sharad Mehrotra[†]
(*mehrotra@csrd.uiuc.edu*)
CSRD and UI Department of Computer Science
1308 West Main Street
Urbana, IL 61801–2307
(217) 244–4657

Luddy Harrison
(*harrison@csrd.uiuc.edu*)
UI Dept. of Computer Science
and
Connected Components Corporation
One Kendall Square, Building 200
Cambridge, MA 02139
(617) 577–1024

1 December 1995

Abstract

In recent work, we described a data prefetch mechanism for pointer-intensive and numeric computations, and presented some aggregate measurements on a suite of benchmarks to quantify its performance potential [MH95]. The basis for this device is a simple classification of memory access patterns in programs that we introduced earlier [HM94]. In this paper we take a close look at two codes from our suite, an English parser called `Link-Gram`, and the circuit simulation program `spice2g6`, and present a detailed analysis of them in the context of our model. Focusing on just two programs allows us to display a wider range of data, and discuss relevant code fragments extracted from their source distributions. Results from this study provide a deeper understanding of our memory access classification scheme, and suggest additional optimizations for future data prefetch mechanisms.

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

[*]Some aspects of this research are covered by a patent application filed by the University of Illinois.
[†]Corresponding author.

Contents

1 Introduction

The ever-increasing gap between microprocessor and memory speeds has been well documented [Gwe92, HP90]. 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[1]. Since secondary caches and main memory are implemented separately from the CPU[2], 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.

Today's advanced processor organizations have achieved a canonical form — a decoupled superscalar implementation. It is well known that the major obstacles to achieving higher performance in processor designs remain control and data hazards [HP90]. Consequently, numerous compiler and architecture research efforts continue to focus on reducing their detrimental effects. Current micro-architectural and compiler practices are most effective at enhancing pipeline core performance. In contrast, most well known data hazard resolution schemes are able to mask latencies of a few cycles at best. The result is that existing processors have imbalanced designs — highly sophisticated pipeline complexes that perform aggresive instruction reordering, but with limited concurrency in their memory interfaces, and little tolerance to long latency data accesses. Modern processors therefore achieve excellent performance on programs whose working sets fit in their primary caches. However, when executing programs with large working sets or complex memory access patterns, CPU performance degenerates, becoming roughly inversely proportional to the number of data cache misses that cannot be masked [CB94]. This problem is particularly severe for load misses, since loads are involved in true data dependences (also known as flow or read-after-write dependences [HP90]), while write misses can often be masked with write buffering.

[1]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.

[2]The DEC Alpha 21164 is an exception, since it integrates a two-level cache on chip.

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[3][CB95, DS95, Gor95, CR94, Mow94, PK94, YGHH94, EV93, JT93, FPJ92, SH92, Sel92, CKP91, 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 [MDO94]. Examples of such computations include transaction processing, graphical user interfaces, text and language processing tools, operating systems, and finite element methods for computational fluid dynamics, structural engineering, etc. Moreover, current compiler technology doesn't offer much assistance in improving the cache performance of loop nests that traverse dynamically linked or sparse data structures. 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.

We have recently described a hardware data prefetch mechanism that is applicable to both pointer-intensive and numeric programs [MH95]. The basis for this device is a simple classification of memory access patterns in programs that we introduced earlier [HM94]. In this paper we take a close look at two codes from our suite, a grammar parser called `Link-Gram`, and the circuit simulation program `spice2g6`, and present a detailed analysis of them in the context of our model. Focusing on just two programs allows us to display a wider range of data, and discuss relevant code fragments extracted

[3]To conserve space, only a selection of references on prefetching techniques has been included.

from their source distributions. Results from this study provide a deeper understanding of our memory access classification scheme, and 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 detailed analysis and 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]. McNiven and Davidson made an early study of the data reference patterns in programs [MD88]. 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 transformations. 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.

In other studies, Vajapeyam and Hsu [VH92] reported basic block characteristics for scalar Cray Y-MP code. Barrett and Zorn [BZ93] measured the lifetimes of dynamically allocated objects to predict lifetimes of short-lived objects, so that they could build better storage allocators. Zorn working in collaboration with Grunwald, has published several papers on other aspects of dynamic memory management in pro-

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

Figure 1: Linear array traversal

grams; see [BZ93] for references. Cvetanovic and Bhandarkar [CB94] used performance monitoring hardware on a DEC Alpha system to produce a thorough instruction-level characterization of that machine running transaction processing and Spec92 workloads. Temam and Jalby have provided a mathematical characterization of the cache behavior of sparse matrix-vector multiplication [TJ92]. Other analytic studies of cache behavior include [FTJ95, LRW91]. The branch prediction and classification problem has received extensive attention in the literature; [YGS95, CHYP94] are two recent studies. Finally, several compiler techniques for memory hierarchy management have been reported, for example [CMT94, BCJ+94, Mow94, Sel92].

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

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

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)

```
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

`*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

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

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

```
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

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

This recurrence can be expressed by the equation

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

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 data structure 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).

So far we have only considered examples where simple data structures are being traversed in loops, and the control flow within the loops is sequential. Next consider

```
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

```
int    n=0;
struct b { int x; double z; struct b* y; };
struct b *p, *q, *r, *s;
...
/* Construct lists with SIZE elts */
q = build_list (SIZE);
r = build_list (SIZE);
...
/* Traverse two lists in a data-dependent manner */
for (p=q, s=r; p!=NULL && s!=NULL; )      /* E */
if (p->x > s->x) {
   n = n + p->x;
   p = p->y;
}
else {
   n = n + s->x;
   s = s->y;
}
```

Figure 5: Conditional linked-list traversal

the contrived loop in Figure 5, which introduces a conditional into the loop body. When loop `E` is executed, it will generate the addresses (ignoring the scalar `n`, and instruction addresses)

```
*q, *r, *q+12, *r+12, *(*q+12) or *(*r+12), *(*q+12)+12, *(*r+12)+12,
    *(*(*q+12)+12) or *(*(*r+12)+12), *(*(*q+12)+12)+12, *(*(*r+12)+12)+12
```

and so on. Two distinct indirect address sequences exist in this address stream; one corresponding to the updates of pointer variable `p`

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

and another corresponding to the updates of pointer variable `s`

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

Both can be described by Equation (2).

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*

Figure 6: The IRB reference prediction table

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` thru `E` of section 3) for prefetching. In this section we briefly describe the IRB; see [MH95] 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.

Figure 6 shows the reference prediction table and the fields that make up an entry in it. The entries in this 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 7 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.

5 Experimental evaluation

In this section we describe the target codes for this study, our experimental framework, and analysis and simulation results.

5.1 Code descriptions

`Link-Gram` Our first program is a system for parsing the English language that is based upon the concept of link grammars [ST91]. A link grammar consists of a set of words, each of which has a linking requirement. A sequence of words can be shown to be a sentence in the language defined by the grammar, if there exists a way to draw arcs connecting the words, so as to satisfy three conditions: (1) The links do not cross, (2) all words in the sequence are connected together, and (3) the links satisfy the linking requirements of each word in the sequence. The linking requirements of all words accepted by the grammar are contained in a dictionary. A dictionary consists of formulas for a set of entries, each of which is for a list of words from the language. These formulas are expressed in terms of connectors, which express the linking requirements of the words in the language, combined by the binary associative operators `and` and `or`.

Figure 7: State transition diagram for the IRB state machine

To simplify the parsing of formulas by mechanical means, the authors use an alternate representation for a link grammar that they call the disjunctive form. In disjunctive form, each word in a grammar has a set of disjuncts associated with it. A disjunct consists of two ordered lists of connector names: the left list and the right list. The disjunct has pointers to the two linked lists of connectors. The disjunctive form representation of a link grammar is parsed with a dynamic programming algorithm that has a $O(N^3)$ running time, where N is the number of words in a sequence being parsed. To speed up the algorithm, the authors apply several techniques, including pruning the number of linkages that need to be checked, a hash-based match data structure that speeds up dictionary searches, and additional simplification rules that take advantage of the ordering requirements of the connectors of a disjunct.

We chose the `Link-Gram` code for this paper several reasons. First, it is a prototypical symbolic code, but one that is not very long (approximately 12,800 lines C code). Second, it uses complex and interesting algorithms. Third, it appears to be quite effective at parsing complex English sentences and capturing several phenomena in the language [ST91]. Finally, it is the basis of several handwriting and voice-recognition tools currently being implemented in industry. We anticipate that such codes will consume ever-increasing numbers of processor cycles in future generation CPUs.

`spice2g6` This is the well known circuit simulation program developed at UC Berkeley. It is capable of analyzing circuits containing resistors, capacitors, inductors, independent and dependent current and voltage sources, transmission lines, and the four most common semiconductor devices: diodes, bipolar junction transistors (BJTs), junction field effect transistors (JFETs), and metal oxide semiconductor (MOS) transistors. Several types of analyses can be performed with `spice2g6`, including nonlinear direct current, nonlinear transient, and linear alternating current analyses. All of the above analysis types can be performed at different temperatures. `spice2g6` contains several built-in models for the four semiconductor devices. Its computational algorithms are iterative and node-voltage oriented, and operate on the modeled circuit's nodal admittance matrix (the Y-matrix). Output can be requested in the form of tables or printer plots. `spice2g6` uses its own internal memory management package.

The `spice2g6` distribution consists of about 18,550 lines of Fortran. The program starts by reading the input file containing the circuit and model parameters provided by the user. As read-in progresses, a set of linked-lists is constructed that contains the circuit being simulated, the device model parameters, etc. The circuit is verified for connectivity once read-in is complete, and this is followed by the main analysis loop, where all the required analyses are performed for each specified temperature. The program finishes up by printing (or plotting) all requested data and simulation run statistics. The principal data structure manipulated by `spice2g6` is the simulated circuit's Y-matrix. This matrix is very sparse[4] and is frequently reordered to minimize fill-in, which involves adding or deleting elements from the Y-matrix after each step of the iterative algorithm. For this reason, the implementors chose a linked-list representation for the Y-matrix. For additional details of the `spice2g6` implementation, consult Chapter 8 of [MA93], and [Coh76].

This version of `spice` is the basis of several commercial re-implementations, and is one of the key tools used by a large portion of the semiconductor industry. Its cache performance becomes progressively worse as larger circuits are analyzed; indeed most engineers rarely simulate circuits of more than a few hundred elements with `spice2g6`. Further, the sparse data structures used in `spice2g6` make it highly resistant to compile-time transformations. For these reasons, it is an interesting code to study.

5.2 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

[4]Because electrical circuits typically only contain elements between a few pairs of nodes, out of the large number possible.

Block Size	Repl. Policy	Cache Size	Associativity	Memory Update Policy	Cache Update Policy	Config Ident.
32 bytes (No sub-blocks)	Least recently used	8K	1 (Direct)	Write through	No write alloc	(1)
				Write back	Write allocate	(2)
			2	Write through	No write alloc	(3)
				Write back	Write allocate	(4)
			4	Write through	No write alloc	(5)
				Write back	Write allocate	(6)
			8 or 256 (Full)	Write through	No write alloc	(7)
				Write back	Write allocate	(8)
		32K	1 (Direct)	Write through	No write alloc	(9)
				Write back	Write allocate	(10)
			2	Write through	No write alloc	(11)
				Write back	Write allocate	(12)
			4	Write through	No write alloc	(13)
				Write back	Write allocate	(14)
			8 or 1024 (Full)	Write through	No write alloc	(15)
				Write back	Write allocate	(16)

Table 1: **Primary data cache design space explored**. Load profiles were built for all 16 cache organizations for each code. Configuration identifiers are used to describe cache configurations in other tables.

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[5]. Table 1 shows the cache organizations for which we built load profiles.

[5]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 will be reported in the first author's forthcoming dissertation.

5.3 Results and analysis

5.3.1 Aggregate data

Table 2 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 2 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[6]. 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[7]. 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 2 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 2. 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

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

[7]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.

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)	# Misses		#LDs causing			# Misses		#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 2: **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 3 and 4.

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.

5.3.2 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 thru with no write allocate write policy. In our opinion, this is a reasonable limiting size for what we expect a practical first level CPU 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 3 and 4 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

Load Id #	Routine Name	Op. Type	How Armed	Execution Count (%age of total)	Linear Count	Indirect Count	Zero Stride	# Read Misses Pre (Cum. %age)	Succ. Predictions (%age lin + ind)	# Read Misses Post
4091	`malloc()`	LW	BOTH+LI	1471087 (1.20%)	44315	1196763	4386	676565 (16.94%)	1121199 (90.34%)	92417(−)
4098	`free()`	LBU	LIN	1371691 (1.12%)	3087	0	19354	365503 (26.09%)	658 (21.32%)	366824(+)
828	`dict_match()`	LB	LIN	1657228 (1.35%)	597735	0	1	149013 (29.82%)	185812 (31.09%)	149366(+)
1466	`mark_dead_connectors()`	LB	LIN	445277 (0.36%)	141	0	0	140649 (33.34%)	18 (12.77%)	141810(+)
1470	`mark_dead_connectors()`	LW	BOTH+LI	420666 (0.34%)	122	592	14	131510 (36.63%)	6 (0.84%)	132764(+)
1763	`copy_Exp()`	LW	LIN	218794 (0.18%)	61895	0	0	116169 (39.54%)	18247 (29.48%)	96453(−)
1753	`size_of_expression()`	LB	LIN	201893 (0.16%)	63	0	0	105040 (42.17%)	9 (14.29%)	105255(+)
1755	`size_of_expression()`	LW	BOTH+LI	191160 (0.16%)	53	293	0	99753 (44.66%)	2 (0.58%)	99997(+)
863	`rabridged_lookup()`	LW	BOTH+LI	401998 (0.33%)	5644	10536	2	94194 (47.02%)	187 (1.16%)	81338(−)
1604	`clean_table()`	LW	BOTH+LI	159163 (0.13%)	94	26	0	87034 (49.20%)	1 (0.83%)	36175(−)
1119	`reverse()`	LW	BOTH+LI	102814 (0.08%)	692	1990	2	83278 (51.29%)	267 (9.96%)	82373(+)
1731	`free_connectors()`	LW	BOTH+LI	195193 (0.16%)	826	4608	42	80326 (53.30%)	1003 (18.46%)	80622(+)
1745	`free_Exp()`	LB	LIN	166066 (0.14%)	604	0	5	71791 (55.09%)	147 (24.34%)	72224(+)
855	`rboolean_dict_lookup()`	LW	BOTH+LI	180512 (0.15%)	3787	4099	795	69559 (56.84%)	0 (0.00%)	58324(−)
1749	`free_E_list()`	LW	LIN	159489 (0.13%)	72	0	4	69470 (58.57%)	0 (0.00%)	69881(+)
1606	`clean_table()`	LH	LIN	159163 (0.13%)	111	0	1	55352 (59.96%)	0 (0.00%)	55661(+)
1770	`copy_E_list()`	LW	NONE	207774 (0.17%)	0	0	0	52860 (61.28%)	0 (0.00%)	40433(−)
1572	`set_dist_fields()`	LW	BOTH+LI	110190 (0.09%)	308	2089	4	47867 (62.48%)	151 (6.30%)	47936(+)
1278	`hash_S()`	LH	LIN	199926 (0.16%)	10672	0	33493	39967 (63.48%)	531 (4.98%)	39927(+)
1280	`hash_S()`	LBU	LIN	199926 (0.16%)	31571	0	11735	38077 (64.44%)	8359 (26.48%)	35883(−)

Table 3: **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 thru with no write allocate write policy. See the text of Section 5.3.2 for a description of the columns, and Table 2 for aggregate characteristics of `Link-Gram`.

involved in any recognizable address sequence.

The fifth field in Tables 3 and 4, *Execution Count*, lists the total execution count for each load, both as the raw count, and as a percentage of the total dynamic count for all load instructions[8]. 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 *# Read Misses Pre*, lists the read misses generated by each load in two different ways. The first number is the actual number of read misses caused by this load for a particular cache configuration (in this case, configuration (11)), while the second figure is a cumulative percentage of read misses caused by this load and all others preceding it in the column, expressed as a fraction of the total read misses (see Table 2). 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 percentage value is the accuracy of prediction with respect to the total number of predictions attempted, which is given by the sum of the linear and indirect counts for the load (i.e. the sum of the values in columns six and seven). 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 3 and 4, we find that approximately

[8]The total dynamic count for each code is listed in Table 2.

Load Id #	Routine Name	Op. Type	How Armed	Execution Count (%age of total)	Linear Count	Indirect Count	Zero Stride	# Read Misses Pre.(Cum. %age)	Succ. Predictions (%age lin + ind)	# Read Misses Post
3124	`dcdcmp()`	LW	BOTH	98101898 (12.66%)	456039	13	385184	10586854 (14.83%)	10644 (2.33%)	10581180(−)
3125	`dcdcmp()`	LW	BOTH	98101898 (12.66%)	392100	7996	385176	9787664 (28.54%)	10644 (2.66%)	9777508(−)
3121	`dcdcmp()`	LW	BOTH	39445586 (5.09%)	88729	13309	397811	8838460 (40.92%)	10649 (10.44%)	8839242(+)
3120	`dcdcmp()`	LW	BOTH	39445586 (5.09%)	74535	3549	397832	7432427 (51.33%)	0 (0.00%)	7435033(+)
6951	`indxx()`	LW	BOTH+LI	7943578 (1.03%)	56175	4273	503	2618788 (55.00%)	16927 (28.00%)	2618503(−)
6950	`indxx()`	LW	BOTH	7943581 (1.03%)	113683	26	493	2576455 (58.61%)	53790 (47.30%)	2566632(+)
3639	`dcsol()`	LWC1	LIN	2366520 (0.31%)	83472	0	0	1686417 (60.97%)	11544 (13.83%)	1675704(−)
3131	`dcdcmp()`	LWC1	LIN	7013424 (0.91%)	130623	0	137517	1443697 (62.99%)	10656 (8.16%)	1438467(−)
3645	`dcsol()`	LW	BOTH	2366520 (0.31%)	83472	12432	0	1322286 (64.84%)	11544 (12.04%)	1317009(−)
3679	`dcsol()`	LWC1	LIN	2362080 (0.30%)	95016	0	0	1254996 (66.60%)	22200 (23.36%)	1246222(−)
3675	`dcsol()`	LW	BOTH	3259848 (0.42%)	15983	4440	0	1113579 (68.16%)	2664 (13.04%)	1112691(−)
3674	`dcsol()`	LW	LIN	3259848 (0.42%)	59495	0	0	973659 (69.53%)	26640 (44.78%)	967231(−)
3646	`dcsol()`	LW	BOTH+LI	2366520 (0.31%)	29304	1777	0	946267 (70.85%)	12432 (40.00%)	943550(−)
3129	`dcdcmp()`	LWC1	LIN	7013424 (0.91%)	239695	0	0	873517 (72.08%)	10656 (4.45%)	866886(−)
3110	`dcdcmp()`	LW	BOTH+LI	7013424 (0.91%)	239695	15101	0	816792 (73.22%)	10656 (4.18%)	806212(−)
3134	`dcdcmp()`	LW	LIN	7013424 (0.91%)	239695	0	0	775383 (74.31%)	10656 (4.45%)	765533(−)
3102	`dcdcmp()`	LWC1	LIN	2366520 (0.31%)	100276	0	0	648061 (75.21%)	23974 (23.91%)	631400(−)
6949	`indxx()`	LW	BOTH+LI	1948098 (0.25%)	829496	6	269	581251 (76.03%)	618134 (74.52%)	507856(−)
2988	`dcdcmp()`	LWC1	LIN	1026437 (0.13%)	24195	0	0	567114 (76.82%)	1609 (6.65%)	565913(−)
3654	`dcsol()`	LWC1	LIN	905760 (0.12%)	91463	0	0	513675 (77.54%)	15984 (17.48%)	501243(−)

Table 4: **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 thru with no write allocate write policy. See the text of Section 5.3.2 for a description of the columns, and Table 2 for aggregate characteristics of `spice2g6`.

ten loads account for 50% of the total read misses for these two codes. Also, from Table 2 it can be observed that if we exclude the direct mapped 8K caches[9], 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 capture more of the cache working set of the program for the given input data set. Looking down the successful predictions columns of Tables 3 and 4, 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 3 and 4. Upon comparing this column with the *# Read Misses Pre* column of Table 3 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 4 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 [MH95].

5.3.3 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 3 and 4. To conserve space, assembly code is only displayed for

[9]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:  }
  [read-dict.c: 830] 0x405fb8: 80830000 lb r3,0(r4)   <== #827
  [read-dict.c: 830] 0x405fbc: 00000000 nop
  [read-dict.c: 829] 0x405fc0: 24a50001 addiu r5,r5,1
  [read-dict.c: 830] 0x405fc4: 10600006 beq r3,r0,0x405fe0
  [read-dict.c: 830] 0x405fc8: 00000000 nop
  [read-dict.c: 830] 0x405fcc: 80a20000 lb r2,0(r5)   <== #828
  [read-dict.c: 830] 0x405fd0: 00000000 nop
  [read-dict.c: 830] 0x405fd4: 1062fff8 beq r3,r2,0x405fb8
  [read-dict.c: 830] 0x405fd8: 24840001 addiu r4,r4,1
  [read-dict.c: 830] 0x405fdc: 2484ffff addiu r4,r4,-1
831:  if ((*s == '*') || (*t == '*'))
832:     return 0;
833:  return (((*s == '.') ? ('\0') : (*s)) - ((*t == '.') ? ('\0') : (*t)));
      ..........
    }
```

Figure 8: MIPS R3000 assembly code for segment from routine `dict_match()` in `Link-Gram`

those statements that either contain the load being studied, or are closely related to it. 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`.

`Link-Gram` From Table 3 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 8 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.

Next, let us examine the routine `mark_dead_connectors()`, which contains LD #1466 (address 0x4097ac) and LD #1470 (address 0x409808). The code for this routine is shown in Figure 9. These loads both have high miss rates and poor predictability, but for different reasons. The operand address for LD #1466 is the contents of register `r16` (a copy of register `r4`). In fact, LD #1466 dereferences a pointer to an `Exp` structure, which is passed in as the first parameter to the routine. Each call to `mark_dead_connectors()` passes in a new `Exp` pointer that has no spatial locality with respect to the previous pointer. This causes numerous cache misses for this load. We get poor predictability for LD #1466 because we do not permit byte loads to participate in indirect sequences[10], and its operand addresses doen't fit a linear address sequence at all. LD #1470 on the other hand, is part of a loop that traverses a linked list. In particular, LD #1471 (address 0x409818) corresponds to updates of the next link pointer, while LD #1470 references a field from the structure being visited. As such, we expect LD #1470 to have poor predictability, because our model applies to loads like LD #1471, not LD #1470. Notice that accurate prediction of LD #1471, will mask misses for LD #1470. Here, LD #1470 is missing heavily because it is the first reference to a field in a new structure.

`spice2g6` First 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 10. 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

[10]Since the contents of byte and halfword loads cannot be used to compose meaningful operand addresses on a MIPS processor.

```
956:int
957:mark_dead_connectors (Exp *e, int dir)
958:{
963:   Connector dummy;
964:   int count;
965:   E_list *l;
966:   dummy.label = NORMAL_LABEL;
967:   dummy.priority = THIN_priority;
968:   count = 0;
969:   if (e->type == CONNECTOR_type) {
  [prune.c: 969] 0x4097ac: 82030000 lb r3,0(r16)  <== #1466
  [prune.c: 969] 0x4097b0: 00000000 nop
  [prune.c: 969] 0x4097b4: 24020002 li r2,2
  [prune.c: 969] 0x4097b8: 1462000f bne r3,r2,0x4097f8
  [prune.c: 969] 0x4097bc: 00008821 move r17,r0
971:       if (e->dir == dir) {
  [prune.c: 971] 0x4097c0: 82020002 lb r2,2(r16)  <== #1467
  [prune.c: 971] 0x4097c4: 00000000 nop
  [prune.c: 971] 0x4097c8: 14520018 bne r2,r18,0x40982c
  [prune.c: 971] 0x4097cc: 02201021 move r2,r17
973:        dummy.string = e->u.string;
  [prune.c: 973] 0x4097d0: 8e020004 lw r2,4(r16)  <== #1468
  [prune.c: 973] 0x4097d4: 00000000 nop
974:        if (!matches_S (&dummy, dir)) {
  [prune.c: 974] 0x4097d8: 27a40010 addiu r4,sp,16
  [prune.c: 974] 0x4097dc: 0c1022d3 jal matches_S
  [prune.c: 974] 0x4097e0: afa2001c sw r2,28(sp)
  [prune.c: 974] 0x4097e4: 14400011 bne r2,r0,0x40982c
  [prune.c: 974] 0x4097e8: 02201021 move r2,r17
976:             e->u.string = NULL;
  [prune.c: 976] 0x4097ec: ae000004 sw r0,4(r16)
977:             count++;
978:        }
979:     }
980:   }
  [prune.c: 980] 0x4097f0: 0810260a j 0x409828
  [prune.c: 980] 0x4097f4: 24110001 li r17,1
981:   else {
983:        for (l = e->u.l; l != NULL; l = l->next) {
  [prune.c: 983] 0x4097f8: 8e100004 lw r16,4(r16)  <== #1469
  [prune.c: 983] 0x4097fc: 00000000 nop
  [prune.c: 983] 0x409800: 1200000a beq r16,r0,0x40982c
  [prune.c: 983] 0x409804: 02201021 move r2,r17
985:            count += mark_dead_connectors (l->e, dir);
  [prune.c: 985] 0x409808: 8e040004 lw r4,4(r16)   <== #1470
  [prune.c: 985] 0x40980c: 00000000 nop
  [prune.c: 985] 0x409810: 0c1025e1 jal mark_dead_connectors
  [prune.c: 985] 0x409814: 02402821 move r5,r18
  [prune.c: 983] 0x409818: 8e100000 lw r16,0(r16)  <== #1471
  [prune.c: 983] 0x40981c: 00000000 nop
  [prune.c: 983] 0x409820: 1600fff9 bne r16,r0,0x409808
  [prune.c: 983] 0x409824: 02228821 addu r17,r17,r2
986:        }
987:   }
988:   return count;
       ..........
  }
```

Figure 9: MIPS R3000 assembly code for segment from routine `mark_dead_connectors()` in `Link-Gram`

```
  ..........
C
C     LOCATE ELEMENT (I,J)
C
  135 IF (J.LT.I) GO TO 145
      LOCIJ=LOCC
  140 LOCIJ=NODPLC(IRPT+LOCIJ)
      IF (NODPLC(IROWNO+LOCIJ).EQ.I) GO TO 155
      GO TO 140
  145 LOCIJ=LOCR
  150 LOCIJ=NODPLC(JCPT+LOCIJ)
      IF (NODPLC(JCOLNO+LOCIJ).EQ.J) GO TO 155
      GO TO 150
  155 VALUE(LVN+LOCIJ)=VALUE(LVN+LOCIJ)-
     1                  VALUE(LVN+LOCC)*VALUE(LVN+LOCR)
  160 LOCC=NODPLC(JCPT+LOCC)
      GO TO 130
  170 LOCR=NODPLC(IRPT+LOCR)
      IF (IPIV.LE.0) GO TO 125
      NODPLC(NUMOFF+I)=NODPLC(NUMOFF+I)-1
      GO TO 125
  ..........
```

Figure 10: Fortran source code for segment containing the heavily missed loads #3120, #3121, #3124, #3125, #3129, #3131, and #3134 in routine `dcdcmp()` from `spice2g6`

elements of `NODPLC` are being accessed with no spatial locality, which explains their poor predictability.

The final routine we consider is `indxx()`, which contributes the fifth and sixth most heavily missed loads, #6950 and #6951. The code for the relevant segment from this routine is shown in Figure 11. Routine `indxx()` takes two arguments, `NODE1` and `NODE2`, that are used to establish the initial entry points into array `NODPLC` whenever `indxx()` is called. Thereafter, both loads are executed about equally. Although the statement on line 36 does resemble Equation (3), it is clear after examining Table 4 that load #6950 is rarely being recognized as an indirect load. Both loads are far more frequently detected as being linear, which is completely dependent upon the Y-matrix node being searched, and the dynamic state of `spice2g6`'s internal memory.

```
          ..........
28: C
29:       N1=NODPLC(IRSWPR+NODE1)
  [indxx.f:  29] 0x4206a0: 00642021 addu r4,r3,r4
  [indxx.f:  29] 0x4206a4: 3c0b1002 lui r11,0x1002
30:       N2=NODPLC(ICSWPR+NODE2)
  [indxx.f:  30] 0x4206a8: 00c52821 addu r5,r6,r5
  [indxx.f:  29] 0x4206ac: 00042080 sll r4,r4,2
  [indxx.f:  29] 0x4206b0: 256b84e0 addiu r11,r11,-31520
  [indxx.f:  30] 0x4206b4: 00052880 sll r5,r5,2
  [indxx.f:  29] 0x4206b8: 01642021 addu r4,r11,r4
  [indxx.f:  30] 0x4206bc: 01652821 addu r5,r11,r5
  [indxx.f:  29] 0x4206c0: 8c84fffc lw r4,-4(r4)    <== #6948
  [indxx.f:  30] 0x4206c4: 8ca5fffc lw r5,-4(r5)    <== #6949
  [indxx.f:  30] 0x4206c8: 00000000 nop
31:       IF (N1-N2) 10,10,30
  [indxx.f:  31] 0x4206cc: 00851023 subu r2,r4,r5
  [indxx.f:  31] 0x4206d0: 1c400014 bgtz r2,0x420724
  [indxx.f:  31] 0x4206d4: 00000000 nop
32: C
33: C     SEARCH COL N2
34: C
35:    10 LOC=N2
  [indxx.f:  35] 0x4206d8: 00056025 or r12,r0,r5
36:    15 LOC=NODPLC(IRPT+LOC)
  [indxx.f:  36] 0x4206dc: 00071025 or r2,r0,r7
  [indxx.f:  36] 0x4206e0: 004c6821 addu r13,r2,r12
  [indxx.f:  36] 0x4206e4: 000d6880 sll r13,r13,2
  [indxx.f:  36] 0x4206e8: 016d6821 addu r13,r11,r13
  [indxx.f:  36] 0x4206ec: 8dacfffc lw r12,-4(r13)  <== #6950
  [indxx.f:  36] 0x4206f0: 00000000 nop
37:       IF (LOC.EQ.0) GO TO 100
  [indxx.f:  37] 0x4206f4: 11800013 beq r12,r0,0x420744
  [indxx.f:  37] 0x4206f8: 00000000 nop
38:       IF (NODPLC(IROWNO+LOC)-N1) 15,20,15
  [indxx.f:  38] 0x4206fc: 010c6821 addu r13,r8,r12
  [indxx.f:  38] 0x420700: 000d6880 sll r13,r13,2
  [indxx.f:  38] 0x420704: 016d6821 addu r13,r11,r13
  [indxx.f:  38] 0x420708: 8dadfffc lw r13,-4(r13)  <== #6951
  [indxx.f:  38] 0x42070c: 00000000 nop
  [indxx.f:  38] 0x420710: 01a46823 subu r13,r13,r4
  [indxx.f:  38] 0x420714: 15a0fff2 bne r13,r0,0x4206e0
  [indxx.f:  38] 0x420718: 00000000 nop
39:    20 INDXX=LOC
40:       RETURN
       ..........
```

Figure 11: MIPS R3000 assembly code for segment containing the heavily missed loads in routine `indxx()` from `spice2g6`

5.4 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[11] 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

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

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.

[BZ93] David A. Barrett and Benjamin G. Zorn. Using Lifetime Predictors to Improve Memory Allocation Performance. In *Proceedings of the SIGPLAN '93 Conference on Program Language Design and Implementation*, pages 187–196, June 1993.

[CB94] Zarka Cvetanovic and Dileep Bhandarkar. Characterization of Alpha AXP Performance Using TP and SPEC Workloads. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 60–70, April 1994.

[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.

[CHYP94] Po-Yung Chang, Eric Hao, Tse-Yu Yeh, and Yale Patt. Branch Classification: a New Mechanism for Improving Branch Predictor Performance. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 22–31, November 1994.

[CKP91] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.

[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.

[Coh76] Ellis Cohen. Program Reference for SPICE2. Technical Report ERL–M592, University of California, Electronics Research Laboratory, Berkeley, CA 94720, 14 June 1976.

[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.

[FTJ95] Christine Fricker, Olivier Temam, and William Jalby. Influence of Cross-Interferences on Blocked Loops: A Case Study with Matrix-Vector Multiply. *ACM Transactions on Programming Languages and Systems*, 17(4):562–575, July 1995.

[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.

[Gwe92] Linley Gwennap. Microprocessor Developments Beyond 1995: Experts See Limits to Superscalar Benefits — Memory Becomes Paramount. *Microprocessor Report*, 9 December 1992.

[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. 08508290, *Prefetch System Applicable to Complex Memory Access Schemes*, filed by the University of Illinois on 27 July 1995.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Mateo, CA 94403, 1990.

[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.

[MA93] G. Massobrio and P. Antognetti. *Semiconductor device modeling with SPICE.* McGraw-Hill, New York, NY, second edition, 1993.

[MD88] Geoffrey D. McNiven and Edward S. Davidson. Analysis of Memory Referencing Behavior For Design of Local Memories. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 56–63, Honolulu, HI, May 1988.

[MDO94] Ann Marie Grizzaffi Maynard, Colette M. Donnelly, and Bret R. Olszewski. Contrasting Characteristics and Cache Performance of Technical and Multi-User Commercial Workloads. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 145–156, October 1994.

[MH95] Sharad Mehrotra and Luddy Harrison. Classifying the performance potential of a data-prefetch mechanism for pointer-intensive and numeric programs. Technical Report 1458, CSRD, University of Illinois at Urbana-Champaign, Urbana, IL 61801, November 1995.

[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.

[ST91] Daniel D. Sleator and Davy Temperley. Parsing English with a Link Grammar. Technical Report CMU-CS-91–196, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA 15213, October 1991. Version 2 of this code has just been released. Check http://bobo.link.cs.cmu.edu/grammar/html/intro.html.

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

[VH92] Sriram Vajapeyam and Wei-Chung Hsu. On the Instruction-Level Characteristics of Scalar Code in Highly-Vectorized Scientific Applications. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 20–28, December 1992.

[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.

[YGS95] Cliff Young, Nicolas Gloy, and Michael D. Smith. A Comparative Analysis of Schemes for Correlated Branch Prediction. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 276–286, June 1995.