An Analysis of Data Access Patterns in Integer Benchmarks

Nikolay Mateev* Paolo Faraboschi† Geoffrey Brown†

November 24, 1997

Abstract

We present memory reference data for the SpecInt95 benchmarks that can be used in evaluating the potential for prefetching spatially correlated data. In contrast with previous work, the data presented are for complete program runs and trace all executed instructions. The data are classified according to their access patterns and we show how types of access patterns arise from the executed benchmark code. Finally, we examine the accuracy of several published beliefs concerning memory access patterns. One common belief is that only scientific code displays sufficiently "regular" behavior for techniques such as prefetching to be of benefit. Our conclusion is that there is significant amount of "regular" behavior even in non-scientific code.

1 Introduction

The goal of this paper is to present memory reference data derived from the SpecInt95 benchmark in order to enable designers to evaluate hardware and software prefetching mechanisms. Prefetching mechanisms attempt to minimize data cache misses by identifying memory reference instructions that display a high degree of spatial correlation and to fetch the data required by such instructions prior to their execution. For a prefetching mechanism to be effective, a significant fraction of executed memory reference instructions must display a strong spatial correlation. The work presented in this paper characterizes the degree of spatial correlation exhibited by the executed memory reference instructions in a significant set of benchmarks.

The amount of data collected from the trace of a significant execution run can be overwhelming. In this paper, we chose to focus exclusively on identifying the spatial correlation for the memory reference instructions of the benchmark programs by median filtering the addresses accessed and computing the average and variance of the stride[1]. Even with such a high degree of compression, each of the benchmarks that we studied has hundreds to thousands of memory reference instructions. Thus, we have developed a simple classification scheme that we use to indicate the types and relative frequencies of memory reference instructions that might be good candidates for prefetching. We also relate high frequency examples of each of the types of reference instructions to their source code. These examples illustrate how the different types of reference instructions arise naturally in a program.

*Department of Computer Science, Cornell University, Ithaca, NY 14853.

†Hewlett-Packard Laboratories, Cambridge, MA 02142.

[1] *Stride* is the address difference between two successive executions of a given memory reference instruction.

The remainder of this paper is organized as follows. In Section 2 we discuss related work. We discuss our target system, compiler, and simulation techniques in Sections 3–5. The simulation workload and results are presented in Section 6. We conclude with a discussion of "common beliefs".

2 Related Work

Because the ultimate goal of memory prefetching is reducing cache misses, published work has often focused upon the effectiveness of various prefetching techniques in reducing cache misses. For example, Charney and Puzak performed a study [2] similar to ours for the SPEC95 benchmarks. There are several differences in our work; the most fundamental is the type of data collected. Charney and Puzak focused upon how cache miss rate changes when varying cache capacity, associativity and line size; limited memory stride data for data cache misses is presented. In contrast, our work is largely independent of an underlying cache model—we present stride data for all memory reference instructions in addition to cache miss data. Furthermore, their data was collected upon only a small part of the execution of the benchmarks and only considered instructions that *missed* at least 500 times while we collected data for entire program runs.

There have been a number of efforts to identify memory reference instructions that are candidates for prefetching within the compiler in order to insert appropriate prefetch instructions. Much of this work has focused upon prefetching loops and scientific code [1, 9, 10, 12]. The underlying assumption of this work is that non-numerical codes are difficult to analyze and generally make good use of cache. More recently some researchers have argued that prefetching non-scientific codes can also be important [6, 8]. Luk and Mowry [8] argue that accessing large data structures causes most cache misses. They present some heuristics for prefetching tree-like data structures, but run into difficulties with hash tables.

There have also been attempts to characterize the likely patterns of memory reference instructions. Selvidge presents argues that the majority of memory references almost never miss, and relatively small number of memory references are often responsible for most of the cache misses [13]. He conjectures that the references with high miss rate are the ones responsible for most of the misses, and cache behavior of individual memory references does not change much with perturbations of input data. Thus he calls the memory references with high miss rate "bad refs", and the ones with low miss rate—"good refs". In theory, prefetching only the bad refs would result in high miss coverage and low overhead. Selvidge's experiments show mixed results and, as we show in Section 7, this is not supported by our data.

3 Target System

For the type of experiments discussed in this paper, system dependent parameters such as the instruction set, memory model, and compiler can significantly impact the results. Our goal in this section is to provide sufficient information for the reader to judge whether the results that we present are likely to reflect those that might be expected on another system. Since our experiments focus upon the data address reference patterns for memory reference instructions, some aspects of our work, such as the fraction of memory reference instructions that have regular interoperation stride, are likely to be stable across a wide range of systems. At the other extreme, results related to cache

miss rates for the various memory reference instructions of a program are sensitive to relatively minor changes in system architecture, but may still be of some value in identifying system design parameters that require careful analysis.

3.1 CPU Architecture

The cpu model used in our experiments is a simulated VLIW [5, 11] processor with a basic RISC instruction set. While our research framework allows us to vary architectural parameters such as the number of functional units and registers, for the experiments discussed here we modeled a machine capable of issuing 8 ALU operations, one memory reference, and one branch per cycle. In addition, we used sufficient registers (32/ALU) to ensure that the memory traffic for register spills and restores is minimized.

It is unlikely that most readers will have direct experience with VLIW machines; however, we argue that the results presented are likely to be similar to those obtained with a superscalar machine with a moderate number of issues/cycle. While the architecture simulated is capable of a large amount of instruction level parallelism (ILP), the code suites that we simulated (e.g. SpecInt95) do not admit highly parallel execution without some amount of code tuning. We did not perform any such code tuning and hence achieved only moderate ILP for the simulations presented. One fundamental difference is that instructions for our architecture are statically scheduled, while for a superscalar they are dynamically scheduled. This difference does not impact the sequence of memory reference instructions accessed—only total number of cycles required to execute a program and hence the effect of data cache misses on overall program performance. The simulated architecture does not incur a large static overhead in code size for unused functional resources.

One limitation of our simulated architecture is a relatively limited set of memory reference instructions that only support a direct register addressing mode. Thus, complex addressing modes such a indexed addressing require additional operations which can typically be bundled into other instructions. While this directly impacts code size, it does not affect the sequence of memory reference instructions issued, and, because of the high potential ILP of our architecture and the relatively powerful instruction scheduler in our compiler, has only limited impact on the total number of instruction cycles required for program execution. Thus, the frequency of memory reference instructions to other instructions, the absolute number of memory reference instructions, and the distribution of data addresses referenced are likely to be identical to a RISC processor.

In addition, our memory system also supports a `prefetch` operation, of the form `prefetch <address_register>`, where the `address_register` has to be explicitly computed. A prefetch operation consumes a memory slot, but the issue width of the machine usually lets the compiler hide the address computation for the value to be prefetched in the previous cycles of the schedule. While the presence of this pref etch operation was the motivation for performing the experiments presented in this paper, the results presented in this paper do not use and hence are not effected by existence of the `prefetch` operation.

3.2 Data Cache

The cache architecture is quite conventional. Our simulated model is an 8KB, two-way set associative, copyback cache with 32B lines and an LRU replacement policy. Our compiler always schedules for a cache hit latency (3 cycles, fully pipelined), and the entire machine stalls if the

cache access misses. Our simulation numbers are somewhat conservative, since we did not simulate a critical-word-first mode and we always expose the copyback penalty.

The penalty to reload a 32B line assumes a 64-bit main memory interface, meaning that we need a burst of four double-words at each refill. Our timing model for the bus is a 6-1-1-1(-2) burst mode, and we assume a ratio of 3 between memory bus speed and processor bus speed. These assumptions yield a refill penalty of 33 processor cycles per miss. A miss that causes a copyback has twice the penalty of a simple miss.

We chose to simulate a rather small cache since to model the interaction between processor and level 1 data caches. However, many of our conclusions are independent of the miss rates for memory reference instructions; we primarily used the relative miss rates as a way to select individual memory reference operations for closer scrutiny. Thus, we believe that most of our results are not particularly sensitive to the size of the data cache and—when properly scaled—can be well generalized for the class of the adopted benchmarks.

4 Compiler technology

Our compiler is a descendant of the Multiflow compiler [4, 7]. The Multiflow compiler uses the *trace scheduling* algorithm [5, 11] to exploit instruction level parallelism across multiple basic blocks. The compiler was originally designed for the Multiflow Trace machine, the first supercomputer-class implementation of the VLIW paradigm [3].

The compiler includes C and Fortran front-ends and implements a set of "traditional" high-level optimization (common-subexpression elimination, if-conversion, loop-unrolling, reductions, etc.). Analysis phases include loop analysis, identification of induction variables, and "disambiguation" of memory aliases.

The trace scheduler identifies "traces" (collections of basic blocks with multiple-entries and multiple-exits) and generates compensation code to patch possible motions of instructions below splits or above joins that may happen in the code generator.

The code generator implements an integrated schedule and register-allocation phase based on list-scheduling heuristics. The register-allocator is a region-based allocator (where the regions are the traces), starts with everything in registers and generates spills when necessary that are added to the DAG and scheduled during the code generation phase itself.

The compiler is robust, generates efficient code, and is flexible enough to be a good research platform. The remainder of our compiler toolchain includes assemblers, linkers, simulators, and analysis tools that allow us to gather precise static and dynamic information on various aspects of the benchmarks and the architecture under evaluation. The results presented in the following sections are an example of the application of this methodology.

5 Simulation Technique

In this section we discuss our simulation techniques including the data collected, the tools used, and the workload that we analyzed.

When performing extensive program simulations, the choice of memory reference data to collect is rather delicate. It is easy to gather too much data—a complete program execution might have

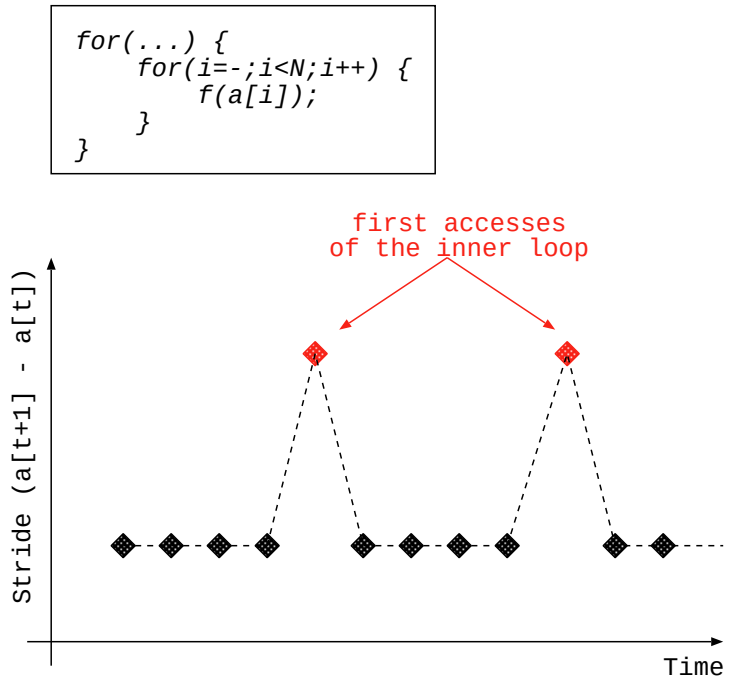

Figure 1: Example of a Typical Stride Pattern for a Nested Loop Access

billions of memory references. Our ultimate research objective was to develop techniques for automating the use of software prefetching. Thus, we focused upon the stride between data addresses for successive executions of a given memory reference instruction. In particular, our simulation environment maintains sufficient state to apply a 3-point median filter to the stride data associated with each memory reference instruction and to compute the mean and variance of the filtered data.

When an access is embedded in deeply nested loops or, in general complex control flow structures, the stride function presents discontinuities that we need to filter out to be able to identify the basic behavior. One measure of "regularity" is for instance a second-order statistic (the variance of the function). However, if we compute the variance of the stride on the unfiltered function, often discontinuities introduce enough noise to make the results meaningless.

For example, we would like to recognize an access within a double-nested loop accessing a large 2-dimensional array as a good candidate with a small and regular stride. However the stride when crossing loop boundaries is quite large and can easily offset the previous measurements. An example of this behavior can be seen in Figure 1.

A 3-point median filter removes simple "spikes" from the stride function. If we see the stride of an access as a function of time (i.e., samples), we would like to be able to capture references to array or similar data structures that exhibit spatial correlation and can benefit from prefetching-like techniques. The basic computation performed for each memory reference is given by the following code fragment (the actual code uses more compact data structures including circular buffers for stride data).

```
reference(ADDRESS data, ADDRESS inst)
{
 int tmp;
 int i = count[inst];

 stride [inst][i] = data - last[inst];
 last[inst] = data;

 tmp = median(stride[inst][i - 2], stride[inst][i - 1],
              stride[inst][i]);
 mean[inst] += tmp;
 meansq[inst] += tmp * tmp;
 count [inst] ++;
 misses[inst] += CacheLookup(data);
}
```

Thus, for each instruction we store the number of executions (`count`), the address of the last data referenced (`last`), the previous two strides (`stride[i-1]` and `stride[i-2]`), the sum of the stride data (`mean`), the sum of the squared stride data (`meansq`), and the cache misses associated with the instruction (`misses`). The purpose of the median filter is to ensure that momentary discontinuities such as those arising at the end of an array row do not dominate the statistics.

Generating meaningful memory access pattern data requires executing actual object code while capturing the data cache behavior of each memory access instruction executed as well as the memory addresses of both the instruction and data. While this can be accomplished using a traditional instruction set simulator, the performance of this approach is likely to be inadequate for extensive simulations. Instead, we use a compiled simulation environment that translates assembly code into C. The compiled simulator (logically) generates a symbolic instruction and memory reference trace. This trace is combined with placement information obtained from native object code to generate an accurate trace of all memory reference instructions. The performance of our simulation system is in excess of 200K simulated instructions/second while gathering the stride data discussed above. In addition, our simulation environment provides the symbolic information necessary to relate each memory reference instruction with the line in the C source from which it was compiled.

6 Simulation Results

The data presented in this paper was generated by simulating a number of integer workloads including the programs in SpecInt95[2], ghostscript 3.33, and gzip. For the SpecInt95 benchmarks, the input data was a "light" version of the reference inputs, for ghostscript the input data was composed of a mix of text, line-art and images, and for gzip the input data was a medium-size file to compress.

Each of the simulation runs was performed on an entire program execution and included roughly 1 billion instructions/run. Table 1 indicates the length of the simulation runs and the number of memory reference instructions executed.

[2]Due to a simulator problem, Perl was not simulated.

	Instructions (millions)	Memory Reference Instructions (millions)
compress	665	162
gcc	1330	302
go	988	247
ijpeg	1800	478
li	1460	343
m88ksim	1030	167
vortex	667	171
gzip	190	39
ghostscript	3140	707

Table 1: Simulation Run Length

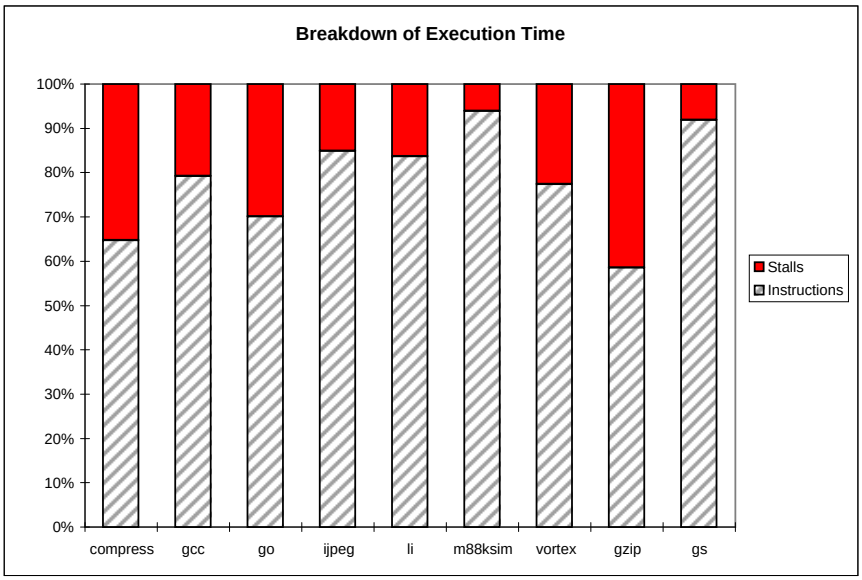

Figure 2: Impact of Cache Misses on Performance

As mentioned, prefetching is used to improve performance by reducing the impact of cache misses. The amount of potential improvement is dependent upon the size and organization of the data cache. For the relatively modest cache discussed in Section 3.2, Figure 2 shows the impact of cache misses upon overall performance. Thus, the potential for improvement due to prefetching ranges from 40% for gzip to 10% for ghostscript (gs).

As mentioned in the Section 5, for each simulation run we computed the mean and variance of the filtered stride data for each memory reference instruction. Based upon this data we developed four categories for the behaviors of memory reference instructions. Those instructions where the mean and variance of the stride data were zero, we call *constant* references. *Regular* memory reference instructions have non-zero mean, but zero variance. *Semi-regular* instructions have non-

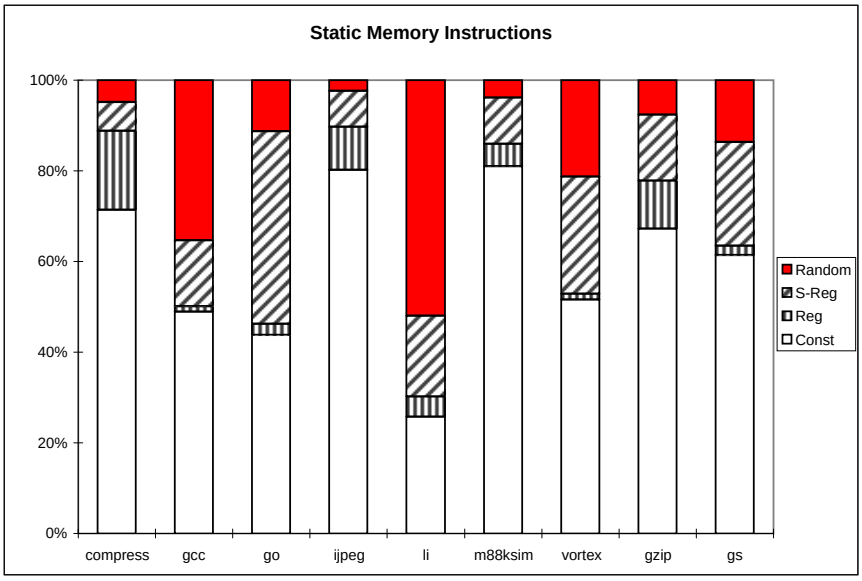

Figure 3: Static

zero mean and "small" variance (less than 10,000—standard deviation of 100). Finally *random* instructions have non-zero mean and "large" variance. In the sequel we give examples of each of these categories of memory reference instructions.

Figures 3, 4, and 5 summarize the relative frequencies of each of the four categories of memory reference instructions for our simulated workload. The relative weight of each type of memory reference instruction is given in three forms. The static distribution is computed by counting each reference instruction in the code once, the dynamic distribution is given by weighting each instruction by the appropriate fraction of reference instructions executed, and the cache miss distribution is computed by weighting each instruction by the fraction of cache misses caused by executing that instruction.

Some observations can be made about the categories of reference instructions simply by knowing what the programs do. For example, ijpeg, a jpeg encoder, performs simple operations that are repeated over an entire image (array)—as expected the majority of memory reference instructions are regular or semi-regular. Similarly, li is a lisp interpreter which performs extensive pointer chasing; the majority of its memory references are random.

Our motivation for performing these simulations was to learn more about the potential for cache prefetching operations. From previously published results, we expect that scientific code would display primarily regular and semi-regular reference patterns and hence would be amenable to prefetching. Similarly we expect random reference patterns to be difficult to prefetch, although as we shall see, not always impossible. In the sequel we present examples of the various types of reference patterns drawn from the code that was simulated. In each case, the instructions presented are among those causing the largest number of cache misses.

For ijpeg most memory references are regular or semi-regular. The following example involves colorspace conversion.

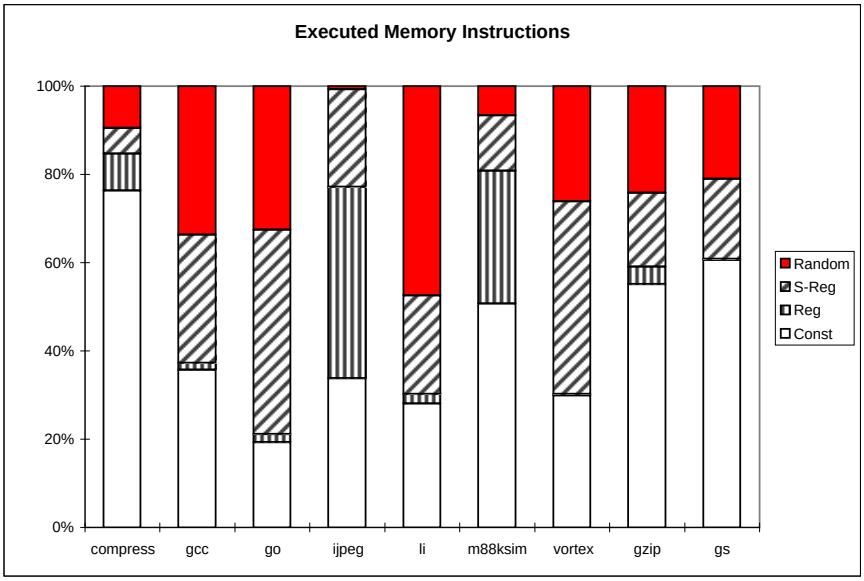

Figure 4: Dynamic

```
while (--num_rows >= 0) {
  ...
  for (col = 0; col < num_cols; col++){
    r = GETJSAMPLE(inptr[RGB_RED]);
    g = GETJSAMPLE(inptr[RGB_GREEN]);
    b = GETJSAMPLE(inptr[RGB_BLUE]);
    ...
    outptr[col] = (JSAMPLE) ((ctab[r+R_Y_OFF] +
                              ctab[g+G_Y_OFF] +
                              ctab[b+B_Y_OFF]) >> SCALEBITS);
    ...
```

The `GETJSAMPLE` references are regular, while the `ctab` references are semi-regular. In this case the regular references arise from iteration over an image, while it appears that the semi-regular references arise from the fact that the rgb data is varies slowly. It seems that this semi-regular data is not amenable to prefetching, while the regular data is.

The program with the largest fraction of cache misses due to random reference instructions is compress. The following code fragment contains the two instructions causing the largest number of misses (`htabof(i)` and `codetabof(i)`):

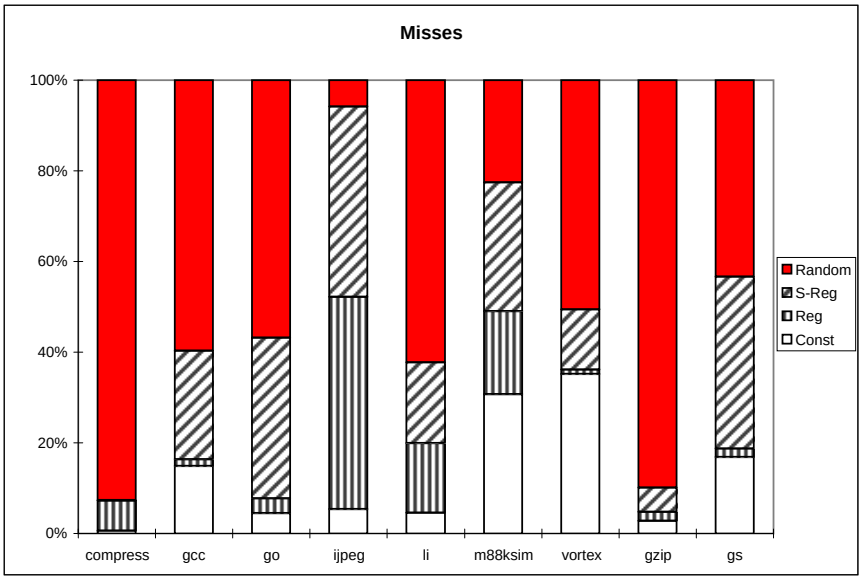

Figure 5: Misses

```
while ... /* 2 lines of code */

  i = ((c << hshift) & ent); /* xor hashing */

  if (htabof (i) == fcode) {
    ent = codetabof (i);
    continue;
  }
```

As can be seen in this example, the random reference pattern arises naturally due to the hash function. In addition, the proximity of the use of the hash value to the `while` statement precludes effective use of prefetching.

Another expected case of random reference patterns is in code that chases pointers. The reference that misses most in li is a pointer used for the mark phase of the garbage collector.

```
if (this->n_flags & MARK) /* most frequent miss */
   break;
else
   /* follow the left sublist if there is one */
   ...
   this = car(prev);
   ...
   /* otherwise, follow the right sublist if there is one */
   ...
   this = cdr(prev);
```

The situation for random reference patterns is not always hopeless. The two instructions in gzip accounting for the largest number of misses also have a random patterns, but the computation of the relevant array index and its use are widely separated in the loop body:

```
do ... /* several lines including a loop */

if (match[best_len]     != scan_end  ||
    match[best_len - 1] != scan_end1 ||

   ... /* several lines including a loop */

len = ...
   ...
best_len = len;
   ...
} while ((cur_match = prev(cur_match & WMASK)) > limit ...);
```

In this case the two instructions (`match[best_len]` and `prev(cur_match & WMASK)`) appear to be viable opportunities for prefetching.

One surprise in our study was the relatively large number of references to fixed addresses causing cache misses. The example from vortex causing the largest number of misses was due resetting a global variable (which appears to be write-only in this program!).

```
Mem_ChunkExpanded = 0;
```

7 Discussion

There are a number of common beliefs concerning memory reference patterns and the likely effects of prefetching. In this section we consider several of these in the context of the data presented in this paper.

Only scientific code needs prefetching. While it is true that scientific code often accesses large data structures and thus causes capacity cache misses, many non-scientific programs also suffer from poor cache behavior. For example, in our experiments gzip spent more than 41% of its running time on memory stalls, followed by compress (35%) and go (30%). On average our benchmarks spent 18% on stalls.

Large data structures cause almost all cache misses. Although large data structures are responsible for many of the cache misses, we observed a surprisingly large number of misses caused by constant-stride memory references (global constants seem to be the main source). More than 35% of the misses in vortex and 30% in m88ksim were of this type. GS and gcc also suffer from constant-stride misses.

Non-scientific code is too hard to prefetch. Clearly array references in innermost loops, which are characteristic for scientific code, are easiest to prefetch. However, our experiments suggest that

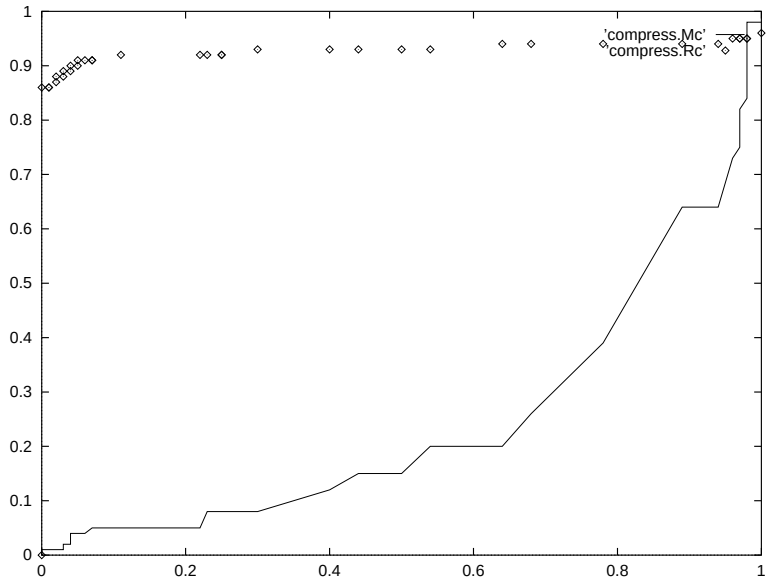

Figure 6: Results: compress

general code might not be as bad as is currently believed. Many programs suffer from constant-stride memory reference misses, which should be easy to eliminate. Also some non-scientific codes exhibits regular access patterns (e.g. ijpeg, m88ksim, li). Moreover, it is likely that minor modifications of existing prefetching techniques would cover a reasonable part of the semi-regular references (24% of the total number of misses in our benchmarks are semi-regular). Not all of the random references are impossible to prefetch, for example, two of the random references in gzip cause more than 60% of all misses, but their addresses can be computed well in advance of their use. Finally, there is often structure behind "random" references. Sparse numerical code is one example for which there are standard techniques [9]. Similarly, encouraging results have been achieved with recursive data structures [8].

"Bad ref" Hypothesis. Our data show that the hypothesis that a small number of references cause most misses is not as universal as suggested in [13]. It is true that most memory references almost never miss. Sometimes, as claimed, high miss rate references cause most of the misses.

This is the case with compress (see Figure 6) and gzip ($R_c(m)$ is the static percentage of memory references with miss rate up to m, and $M_c(m)$ is the percentage of misses caused by memory references with miss rate up to m). Unfortunately, often memory references with low miss rates are responsible for the majority of misses; an extreme example is ijpeg (Figure 7). The behavior of gcc (Figure 8) is representative for the rest of our benchmarks.

In summary, while conventional wisdom holds that only scientific code displays sufficiently regular memory access patterns for techniques such as prefetching to be useful, our data shows that the programs in the standard integer benchmarks display significant amounts of regular accesses.

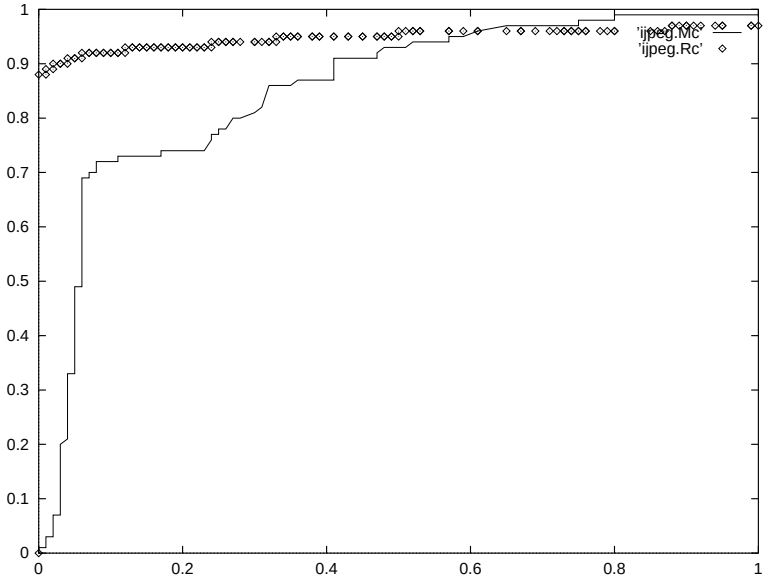

Figure 7: Results: ijpeg

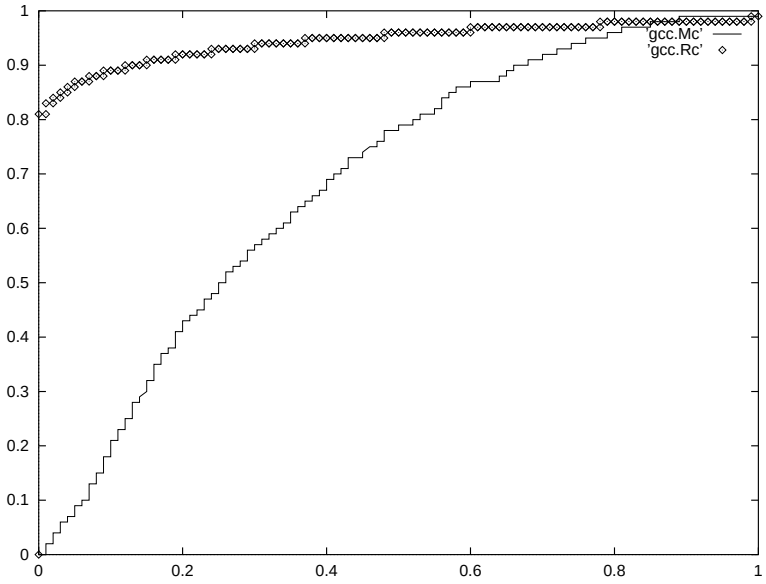

Figure 8: Results: gcc

References

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

[2] M. J. Charney and T. R. Puzak. Prefetching and memory system behavior of the SPEC95 benchmark suite. *IBM Journal of Research & Development*, 41(3), 1997.

[3] Robert P. Colwell, Robert P. Nix, John J. O'Donnell, David B. Papworth, and Paul K. Rodman. A VLIW architecture for a trace scheduling compiler. In *Proceedings Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II) in SIGPLAN Notices*, pages 180–192, Palo Alto, CA, October 1987.

[4] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. MIT Press, Cambridge, 1986.

[5] Joseph A. Fisher. Very long instruction word architectures and the ELI-512. In *Tenth Annual International Symposium on Computer Architecture*, pages 140–150, Stockholm, Sweden, June 1983. IEEE Computer Society.

[6] Mikko H. Lipasti, William J. Schmidt, Steven R. Kunkel, and Robert R. Roediger. SPAID: Software prefetching in pointer- and call-intesive environments. In *The 28th Annual International Symposium on Microarchitecture*, pages 231–236, 1995.

[7] P. Geoffrey Lowney, Stefan M. Freudenberger, Thomas J. Karzes, W. D. Lichtenstein, Robert P. Nix, John S. O'Donnell, and John C. Ruttenberg. The Multiflow Trace Scheduling compiler. *The Journal of Supercomputing*, 7(1/2):51–142, May 1993.

[8] Chi-Keung Luk and Todd C. Mowry. Compiler-based prefetching for recursive data structures. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, September 1996.

[9] Todd C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, March 1994.

[10] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, October 1992.

[11] B. Ramakrishna Rau and Joseph A. Fisher. Instruction-level parallel processing: History, overview, and perspective. *The Journal of Supercomputing*, 7(1/2):9–50, May 1993.

[12] Vatsa Santhanam, Edward H. Gornish, and Wei-Chung Hsu. Data prefetching on the HP PA-8000. In *The 24th Annual International Symposium on Computer Architecture*, pages 264–273, 1997.

[13] Charles Selvidge. *Compilation-Based Prefetching for Memory Latency Tolerance*. PhD thesis, MIT, May 1992.