Abstract

Since a program may have an infinite number of inputs, it is difficult to measure the exact performance under all possible uses. In this paper, we use the notion of reuse distance and reuse signature to measure the variation of data locality across program inputs. We use two classes of programs. The first is integer programs from SPEC 2000 benchmark set, including Vpr, Mcf, Parser, Perl, Gzip, and Bzip. For this set we test the current version from their open-source web sites instead of (or in addition to) the ones in the benchmark set. The second class is commonly used interactive programs, including Latex, Ghostview, and Gnuplot. We compare the reuse signature of 126 inputs for these 9 programs and show different degrees of locality variation. For most programs, the reuse signature is mostly consistent across inputs, indicating that the reuse pattern can be exploited in system design and optimization.

* The figures in this paper are generated by pdflatex and can be printed clearly on paper even though they may not display well on screen.

1 Introduction

Caching is widely used in general and special purpose computer systems, and cache performance increasingly determines system speed, cost, and energy usage. The effect of caching depends on program locality, which changes from program to program and also from input to input for the same program. In this paper, we study the locality variation caused by the difference in program inputs.

Many factors may change the locality for the same program. Different inputs may generate a different amount of data and cause a different length of computation. The command line flags may cause a program to invoke completely different modules. For interactive programs such as an interpreter, the control flow and data allocation are determined by the input. While in principle one can construct a program that has no consistent behavior across different inputs, in reality many commonly used programs do not behave randomly. The question is then an experimental one: how much locality variation exist in a given program.

Historically, except for the most frequently executed code and accessed global variables, it was difficult to correlate behavior across different runs of the same program. If most data are dynamically allocated, the correlation becomes
especially difficult because the same data may be allocated in different locations in different runs. One solution is to use reuse distance and reuse signature patterns, which are defined in the next section. Our earlier work examined the locality variation for 13 floating-point and integer applications in SPEC and NAS benchmark sets [7]. Due to the limited inputs provided by the benchmark distribution, the past study used only three inputs for most programs and could not analyze a number of complex integer programs due to insufficient inputs as well as limited computing resources at the time.

In this work, we examine the locality variation for two classes of programs. The first is integer programs from SPEC 2000 benchmark set that were not studied in our earlier paper. We pick programs that have an open-source implementation and have practically an unlimited number of inputs available. These include two solvers (Mcf and Vpr), two compression programs (Gzip and Bzip), an English parser (Parser), and the Perl interpreter.

To understand input-driven programs, we use three commonly used interactive programs—Latex, Ghostview, and Gnuplot, which also have practically an unlimited number of inputs. The interactive programs are more important to embedded systems such as Palm Pilots, cellular phones, and MP3 players because traditional SPEC CPU benchmarks are not run on these systems. Ultimately, we compare the reuse signature of 126 inputs for these 9 programs, performing locality analysis on traces having a cumulative length of over 250 billion accesses.

In this paper, we measure only the whole-program locality, which is an average across all data and the entire execution. Reuse distance can be used to examine finer temporal and spatial patterns, but such is outside the scope of this paper. The whole-program locality is useful in memory hierarchy design, program optimization, and locality-aware job scheduling. We measure the reuse pattern at the finest granularity—each distinct memory address is a data element. In other words, we measure the temporal locality not the spatial locality. Temporal locality, especially long distance reuses, reveals large-scale properties of complex programs. Spatial locality, on the other hand, depends on not only cache implementation such as cache block sizes and cache associativity but also program implementation such as data placement. Our earlier studies showed that predictable temporal locality patterns lead to predictable cache locality patterns [22]. We expect to repeat similar tests in the near future.

The remainder of this paper defines reuse distance, discusses the reuse distance histograms of our instrumented programs, extends into other applications of reuse distance analysis and concludes with a final summary.
2 Reuse Signature Pattern

By ordering the memory accesses of a program execution by logical time, we obtain a program trace. Then, *reuse distance* is the number of distinct data elements accessed between two consecutive uses of the same element. The distribution of all reuse distances in a trace is its reuse distance histogram.

In this work, we create the histogram by partitioning the range of the reuse distances into bins and counting each time a reuse distance falls into its corresponding bin. Because large inputs can distort the frequencies on smaller inputs, we scale each input by the total number of memory accesses for that input. This allows us to properly compare each input via percentage reuse distance histograms. For brevity, we shorten the phrase ‘percentage reuse distance histogram of a trace’ to *reuse signature*. By extension, the *reuse signature pattern* of a program is the pattern of reuse signatures across all inputs.

For clarification, refer to Figure 1. In the example trace shown in Part (a), the reuse distance of the second access to \( b \) is 2 because two distinct elements, \( a \) and \( c \), are accessed between this and the first access to \( b \). To calculate the reuse signature, we measure the reuse distance for each access in the trace. Following the convention introduced by Beyls [2], we set the reuse distance of the first access of a data element to infinity. The reuse signature, shown in Figure 1(b), is the histogram of all finite reuse distances. It shows, for example, that 50% of reuses have a distance of 2.

![Figure 1: Reuse distance and reuse signature](image)

The reuse signature of a trace gives the hit/miss rate for fully associative cache of all sizes [13]. It can also be used to accurately estimate the hit/miss rate for direct mapped or set-associative caches [12, 16, 22]. The reuse signature is a property of the trace and is independent of hardware parameters. It is mostly insensitive to the coding style of a program. For example, unrolling the innermost loop or changing the allocation order of data variables does not change the reuse signature of an execution.
Efficiency is a problem in measuring long distance reuses. A number of algorithms have been developed over the past thirty years [1, 10, 13, 14, 19]. Our earlier work developed an algorithm that only stores approximate reuse distances. By trading accuracy over efficiency, we reduce the runtime cost to $O(N \log \log M)$ and the space cost to $O(\log M)$, where $N$ is the length of the program trace and $M$ is the size of the program input [7]. In this work, we use the analysis having 99.9% accuracy, meaning that the measured reuse distance is guaranteed to be between 99.9% and 100% of the actual distance.

3 Methodology

Pre-existing code used by [7, 15, 21, 22] have instrumented Fortran and C code under Alpha machines using the ATOM instrumentor [18]. We port their code to run under x86 machines and instrument using Intel’s Pin tool [11]. Specifically, we compile using gcc version 3.2.2 on Linux kernel 2.4.20 running on Intel Pentium 4 2.0 GHz machines. The only exception is the SPEC version of Gzip, which was instrumented with ATOM on a Sun Alpha machine with four 599 MHz processors.

Our test programs primarily come from the SPEC 2000 Integer benchmark suite, but also include various other programs commonly found on Linux distributions (refer to Figure 2). We have tested all SPEC 2000 Integer programs but do not include Eon, for lack of up-to-date source code; Crafty, for lack of inputs; and Gap, for difficulties in running even the provided inputs. Since the SPEC 2000 benchmarks consist of slightly modified code (for benchmarking purpose), we obtain the original version of each program from each corresponding author’s referenced web page.

<table>
<thead>
<tr>
<th>Program</th>
<th>Origin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bzip</td>
<td>SPEC 2000</td>
<td>Compression utility</td>
</tr>
<tr>
<td>Gzip</td>
<td>SPEC 2000</td>
<td>Compression utility</td>
</tr>
<tr>
<td>Mcf</td>
<td>SPEC 2000</td>
<td>Network simplex implementation</td>
</tr>
<tr>
<td>Parser</td>
<td>SPEC 2000</td>
<td>English grammar Parser</td>
</tr>
<tr>
<td>Perl</td>
<td>SPEC 2000</td>
<td>Programming language</td>
</tr>
<tr>
<td>Vpr</td>
<td>SPEC 2000</td>
<td>FPGA placement and routing tool</td>
</tr>
<tr>
<td>Gnuplot</td>
<td>Linux tool</td>
<td>Math plot utility</td>
</tr>
<tr>
<td>Gv</td>
<td>Linux tool</td>
<td>Postscript/PDF file viewer</td>
</tr>
<tr>
<td>Latex</td>
<td>Linux tool</td>
<td>Document preparation system</td>
</tr>
</tbody>
</table>

Figure 2: Description of Instrumented Programs
4 Experimental Results

4.1 Latex

Recall that compiling a Latex document having a bibliography is a four stage process involving a stage 1 Latex command, a stage 2 Bibtex command, and two more stage 3 and 4 Latex commands. Figure 3 summarizes a description of the inputs for Latex, including the stage number at which each command was issued. Since Latex is most often used in scientific writing, the representative inputs consist of a conference paper, three professional documents, and a doctorate thesis. Also, there is one sample document that utilizes several Latex features including formulas, tables, and pictures.

Figure 13(a) clearly shows that the reuse signature pattern of Latex is constant. This is expected since Latex is a compiler. Compilers have highly structured input grammars and have very methodical internal algorithms. Unvaried reuse signature patterns are typical of compilers.

4.2 Vpr

Since Vpr is an FPGA placement and routing tool, the representative inputs are various logic functions. Figure 4 details the number of inputs, outputs, and blocks for each logic function. Consider the block count as an estimate of how large that particular function is. Vpr also has a simulation feature, which was suppressed with the \texttt{-route\_only} option.

Like Latex, Vpr also has a constant reuse signature pattern (see Figure 13(b)). This indicates that Vpr’s routing algorithm behaves independently of differences between logic functions.

<table>
<thead>
<tr>
<th>Input Name</th>
<th>Page Count</th>
<th>Stage Latex was invoked</th>
</tr>
</thead>
<tbody>
<tr>
<td>03lcpc</td>
<td>15</td>
<td>Stage 1</td>
</tr>
<tr>
<td>Hierarchy</td>
<td>15</td>
<td>Stage 2</td>
</tr>
<tr>
<td>Prediction</td>
<td>26</td>
<td>Stage 1</td>
</tr>
<tr>
<td>cdThesis</td>
<td>117</td>
<td>Stage 1</td>
</tr>
<tr>
<td>gzippaper</td>
<td>6</td>
<td>Stage 1</td>
</tr>
<tr>
<td>wewSample</td>
<td>7</td>
<td>Stage 3</td>
</tr>
</tbody>
</table>

Figure 3: Description of Latex inputs
<table>
<thead>
<tr>
<th>Input Name</th>
<th>Inputs/Outputs</th>
<th>Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>alu4</td>
<td>14/8</td>
<td>1544</td>
</tr>
<tr>
<td>apex2</td>
<td>38/3</td>
<td>1919</td>
</tr>
<tr>
<td>apex4</td>
<td>9/19</td>
<td>1290</td>
</tr>
<tr>
<td>des</td>
<td>256/245</td>
<td>2092</td>
</tr>
<tr>
<td>ex1010</td>
<td>10/10</td>
<td>4618</td>
</tr>
<tr>
<td>ex5p</td>
<td>8/63</td>
<td>1135</td>
</tr>
<tr>
<td>misex3</td>
<td>14/14</td>
<td>1425</td>
</tr>
<tr>
<td>pdc</td>
<td>16/40</td>
<td>4631</td>
</tr>
<tr>
<td>seq</td>
<td>41/35</td>
<td>1826</td>
</tr>
<tr>
<td>spla</td>
<td>16/46</td>
<td>3752</td>
</tr>
</tbody>
</table>

Figure 4: Description of Vpr inputs

<table>
<thead>
<tr>
<th>Input Name</th>
<th>Input file</th>
<th>Nodes</th>
<th>Arcs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>gte_bad.40</td>
<td>49</td>
<td>520</td>
</tr>
<tr>
<td>2</td>
<td>gte_bad.6830</td>
<td>49</td>
<td>520</td>
</tr>
<tr>
<td>3</td>
<td>gte_bad.508829</td>
<td>49</td>
<td>520</td>
</tr>
<tr>
<td>5</td>
<td>input</td>
<td>100</td>
<td>150</td>
</tr>
<tr>
<td>6</td>
<td>output.txt</td>
<td>15</td>
<td>91</td>
</tr>
</tbody>
</table>

Figure 5: Description of Mcf inputs

4.3 Mcf

The representative inputs to Mcf, a network simplex implementation that finds the minimum cost flow of a graph, consist of 3 different graph topologies. One topology, gte_bad, has been duplicated 3 times with different capacities. Refer to Figure 5 for details.

Figure 13(c) shows that Mcf has one trend with some variability. Also note that all the reuse signatures based upon gte_bad is identical. This indicates that the reuse signature pattern for Mcf is completely independent of graph capacity and only slightly dependent on graph topology.

4.4 Parser

Parser takes in English sentences and outputs their grammatical sentence structure. As detailed in Figure 6, we chose representative inputs of professional and unprofessional quality, varying document length, and spoken/written language styles.

The reuse signature pattern for Parser in Figure 13(d) shows that the program has one trend that has some variability. Although English grammar is pretty structured, idiosyncrasies are frequent. Such irregularities could affect the variability in the reuse signature pattern of Parser.
4.5 Gv

Because Gv reads postscript and PDF formats, our inputs (Figure 7) sample both formats. Additionally, we vary document length and file composition. Variable compositions include presence of color, pictures, tables, or equations. Also note that Gv is an interactive program. As such, it also requires user input in addition to the file input. Since Gv is a viewing utility, our interactive input consisted only of scrolling through the entire length of each document.

Figure 13(e) shows that the reuse signature pattern for Gv has two trends that differ significantly only at bin 6143. On closer inspection of the program inputs, observe that one trend is for postscript input files and the other is for PDF input files. Thus, the fact that Gv behaves differently to two input formats is expected.

4.6 Perl

Although Perl is a highly rich programming language, its actual usage is mostly limited to parsing text. Thus, as shown in Figure 8, our inputs include four text processing scripts. The representative inputs also consist of two trivial programs and two number computing scripts.

The reuse signature pattern for Perl in Figure 13(f) has one trend with medium variability. Considering the fact that program behavior could be highly variable and Perl is a programming language, it is surprising that the reuse pattern for Perl is as unvaried as it is. This suggests that the interpreter itself is imposing the reuse signature pattern.
### Input Name | Description
--- | ---
awstats-6.4 | Calculates web statistics from web logs
calcPerHist | Calculates percentage reuse distance histograms from raw data
empty | Empty script that does nothing
helloworld | Prints out hello world
parseAllData | Accumulates data from several files into one file
perfect | Calculates perfect numbers
pi | Calculates pi in four different ways
plotPerHist | Outputs Gnuplot script that plots all percentage reuse distance of a program

Figure 8: Description of Perl inputs

### Input Name | Description
--- | ---
airfoil | 3D generated model of airplane wing
airfoil2PNG | Same as airfoil but outputs to PNG file
butterfly | Example of polar coordinates
butterfly2PNG | Same as butterfly but outputs to PNG file
color | Example of color and occlusion
fancyscript | Example of text, special symbols, and justification
fancyscript2PNG | Same as fancyscript but outputs to PNG file
glass | Example of 3D data grid plotting
glass2PNG | Same as glass but outputs to PNG file
jett | Example of piecewise function plotting
multiplot | Example of having multiple plots
perHist | Example of 2D data grid plotting
random3d | Example of random 3D plotting
rosenbrock | Example of 3D function plotting
whale | 3D data mesh model of whale
world | Example of 2D projection on spherical coordinates

Figure 9: Description of Gnuplot inputs

### 4.7 Gnuplot

Because Gnuplot is a highly versatile plotting utility, our representative inputs must plot data points or functions in 2 or 3 dimensions in different coordinate systems (refer to Figure 9).

Figure 13(g) shows that the reuse signature pattern of Gnuplot is highly variable. Although many reuse signatures follow a trend at low bin numbers, several signatures do not, particularly at higher bin numbers. Nevertheless, one consistent pattern is the trend from Gnuplot scripts that output to PNG files. This behavior indicates that Gnuplot is invoking a large external call when exporting to PNG. The trend of these reuse signatures reflects the reuse signature pattern of the external call.
4.8 Bzip

Because Bzip is a general purpose compression utility, its representative inputs must have a variety of file sizes, file types, and compressibility. Thus, our inputs include text, binary, audio, video, and picture files of up to three different sizes (refer to Figure 12).

As shown in Figure 13(h), the reuse signature pattern of Bzip is highly variable. This demonstrates that not all programs have input independent program behavior.

4.9 Gzip

We tested Gzip with the same set of inputs as Bzip. Unlike Bzip, the reuse signatures from Gzip were much more regular (see Figure 10). Inputs tend to follow one of two trends, which are distinguished by either a peak or slope at bin 31. Only the zero input defies both trends. However, this input is atypical for Gzip and was included only to test extreme cases of input.

It is interesting to note here that Cheng and Vayani instrumented the original SPEC 2000 Gzip benchmark using ATOM on a Sun Alpha machine. They determined that the reuse signatures in Gzip (see Figure 11) followed one of three possible trends, which they labeled as the small file, text file, and multimedia trends [4]. These trends reflect the underlying Lempel-Ziv compression algorithm of Gzip. As such, the SPEC 2000 version of Gzip is the prime example of the potential that reuse signatures have in capturing program behavior.
Figure 11: Reuse signature of SPEC 2000 Gzip
5 Applications of Reuse Signature Pattern

The reuse signature pattern is useful for at least three reasons. The first is cache design in a general or special purpose processor. A designer needs to pick the cache configuration that satisfies space, energy, and cost constraints while minimizing the number of cache misses for representative workloads. The reuse signature patterns do not have timing information, which is needed for measuring the exact performance impact. Modern processor design relies on detailed processor simulation. Since cycle-accurate simulation is time consuming, it is usually applied to a set of representative programs, each with a few inputs. While various sampling schemes can reduce the time by simulating partial program traces, they do not guarantee the accuracy of the results for an unknown input. In this paper, we model the locality variation across a wide range of inputs for applications that have input dependent behavior such as compilers, interpreters, and other interactive programs. The information can help to choose representative inputs for these programs and to evaluate the cache design without performing a full CPU simulation.

The reuse signature pattern can also be used to calculate program balance between computation and memory needs. Program balance was originally defined as the average amount of memory loads needed for each calculation [3] and later extended to include multi-level cache hierarchies [6]. Program balance is used in embedded system design to direct the allocation of resources between computation and memory functions [17]. Earlier studies use compiler
Figure 13: Reuse signatures. (Number of data elements : Number of Accesses) follow input labels.
analysis for regular loop nests and profiling for whole executions. However, profiling does not measure how the balance changes across different inputs. The reuse signature pattern shows the change in the memory need of a program. If the computing need can be modeled across program inputs, the balance of dynamic programs can then be used to help better design embedded systems such as Palm Pilots and cellular phones for running dynamic, interactive programs.

Finally, the reuse signature pattern can be used in scheduling of applications on a heterogeneous environment with different machines installed with a different cache hierarchy. The pattern can be used to predict the miss rate on any number of cache levels of different configurations. A scheduler can then use the prediction for selecting machine-program pairs, in addition to considering the CPU speed and the system load on the machine. Since a scheduled application may take an unknown input, it is useful to know the possible range of the locality behavior.

6 Related Work

Wall presented an early study of execution frequency across multiple runs [20]. Chilimbi examined the consistency of frequent data access streams [5]. Since data may be different from one input to another, Chilimbi used the instruction PC instead of the identity of data and found that hot streams include similar sets of instructions if not the same sequence. Hsu et al. compared frequency and path profiles in different runs and found that small inputs often had different behavior than larger inputs [9]. Eeckhout et al. studied correlation in 79 inputs of 9 programs using principal components analysis followed by hierarchical clustering[8]. They considered data properties including access frequency of global variables and the cache miss rate.

This work measures the variation of the reuse signature of different inputs of a program. It leverages the techniques used by others, especially the statistical clustering by Eeckhout et al. By profiling a large number of inputs, it shows common patterns in integer benchmarks and interactive programs.

This is an extension of our earlier study on scientific and integer programs. The past study modeled the constant, linear, and sub-linear patterns and allowed prediction of the reuse signature for unknown program inputs, including those that are too large to run, let alone to simulate [7]. Here we examine the presence of such patterns in a larger set of programs and show new patterns that cannot be precisely characterized by old methods.
7 Summary

The reuse signatures of Latex and Vpr were constant across all inputs. For Mcf and Parser, there was a single trend having some variability. Signatures of Gv followed two expected trends: one for postscript and another for PDF files. Signatures of Perl and open source Gzip had trends with medium variability. Gnuplot and Bzip both had trends with high variabilities. Since seven of the nine instrumented programs had consistent reuse signatures, we can expect to use these signatures in system design and optimization for many more programs.

8 Acknowledgements

The research is supported mainly by an REU grant from NSF and partially by the Department of Energy (Contract No. DE-FG02-02ER25525) and the National Science Foundation (Contract No. CCR-0238176, CNS-0509270, and EIA-0080124). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding organizations.

References


