Lossless Trace Compression

Eric E. Johnson, Jiheng Ha, and M. Baqar Zaidi

New Mexico State University

Abstract

The tremendous storage space required for a useful data base of program traces has prompted a search for trace reduction techniques. In this paper we discuss a range of information-lossless address and instruction trace compression schemes that can reduce both storage space and access time by an order of magnitude or more, without discarding either references or inter-reference timing information from the original trace.

The PDATS family of trace compression techniques achieves trace coding densities of about six references per byte. This family of techniques is now in use as the standard in the NMSU TraceBase, an extensive trace archive that has been established for use by the international research and teaching community.

Index Terms: Trace reduction, trace compression, lossless coding, trace-driven simulation.

1. Introduction

For those seeking to understand the behavior of complex systems, simulation is often the tool of choice. Each of the various components of the system under study can be simulated at a level of abstraction appropriate to the degree of interest in its behavior. Simplifying assumptions need be introduced only where dictated by lack of knowledge or of simulation resources. New system configurations can be evaluated quickly merely by changing the parameters of the simulation model of the system.

The accuracy of simulation results depends upon the fidelity of the model of the system as well as of the workload applied to the simulated system. In some cases, the workload can be modeled with sufficient fidelity by streams of random numbers having carefully crafted distributions. In other cases, workloads contain patterns of varying scales that can be difficult to accurately model

with random number streams; in such cases, simulations are often driven by traces of workloads.

A trace is an ordered sequence of events, sometimes including event timing. Two types of traces are commonly used in the performance evaluation of computer subsystems:

- *Memory hierarchy* simulations (e.g., cache memories) are driven by *address traces*.
- *CPU* simulations, which include the simulation of pipelined ALUs, load-store units, and other functional units, are driven by *address and instruction traces*.

1.1 Trace Size

In general, the metric of interest in simulating a computer is the time that that computer will require to execute some program of interest. Programs will usually be of interest only if they require execution times that are noticeable by humans, i.e., times greater than about 100 ms. With CPU speeds currently on the order of 100 million instructions per second (MIPS), programs of interest will often involve the execution of billions of instructions. Traces of the complete execution of these programs will therefore contain billions of memory references.

Furthermore, evaluation of a computer design using only one trace is seldom sufficient. General-purpose computers are expected to exhibit reasonably consistent performance over a wide range of workloads. Therefore, the evaluation of a computer system using trace-driven simulation must comprise a series of simulations using traces of programs that cover the range of interesting applications of that system. Thus, we find that performance evaluation of computer systems using trace-driven simulation requires a collection of traces, each of which includes millions or billions of references.

The disk (or tape) space required for storage of such a trace archive is the product of the number of references stored in the trace archive and the number of bytes required to store each reference. When traces are stored in the usual ASCII format (dinero [1]), references in address traces typically consume 8 bytes each, while address and instruction traces require about 16 bytes per reference. Thus, file sizes of 1 GB each are not unusual for medium-sized traces of 10 to 100 million

references, and it is quite easy for a small library of traces to consume many gigabytes of secondary storage unless the number of bytes per reference is somehow reduced.

Trace reduction techniques reported in the literature include filtering [2-6], sampling [7], and compression [8-9]. Trace filtering discards all references that hit in a small "filter cache." Trace sampling stores relatively short bursts of references (e.g., 1 million) at regular intervals, discarding the intervening references. Only trace compression techniques retain all of the information from the trace. Such techniques are the topic of this paper.

A related topic is production and consumption of traces "on the fly" without the necessity of ever storing the complete trace. This can be an attractive alternative when traces can be exactly regenerated for each alternative to be simulated. However, it can be problematic to exactly regenerate traces of some systems, including those that interact with the external environment.

1.2 Simulation Time

Another consideration in trace-driven simulation is the time required to read the references from the trace file. Reducing the trace file size results in less data movement from secondary storage to main memory, and therefore less time required for file I/O in a trace-driven simulation. However, this time savings comes at a cost of CPU time to decompress the trace, so the net effect on simulation time depends upon the complexity of the decompression algorithm.

1.3 Overview of the Paper

The following sections discuss the characteristics of program traces that permit effective, lossless compression, followed by presentation of two lossless trace compression techniques: the PDATS address trace compression scheme, and the PDI scheme which extends the PDATS technique to compressing address-and-instruction traces.

2. Redundancy in Address Traces

The goal of trace compression, as opposed to trace filtering or sampling, is to reduce the size of trace files without discarding any of the information contained in the trace. Compression must therefore remove only redundancy from the trace. Fortunately, the reference streams captured in address traces and the ASCII record format usually used to encode them are both highly redundant.

2.1 Reference Stream Redundancy

Beginning with the reference stream itself, it is well known that the address sequences generated by processors running common applications exhibit both spatial and temporal locality. Instruction streams are also strongly sequential. This section presents a formal model of spatial locality, along with measurements of spatial locality and sequentiality from a range of traces.

Spatial Locality

The sequence of addresses produced by a processor during the execution of a program may be considered as a composite of several types, or streams, of references: instruction fetches, accesses to the stack, to the heap, and so on. Although the differences in address from one reference to the next sequential reference may appear somewhat chaotic, when a trace is resolved into its constituent components we find fairly stable and local reference patterns within each stream.

Spatial locality means that the next reference generated is usually quite close to the previous reference from the same stream. Thus, the differences in address from one reference to the next in that stream are likely to be small. Mathematically, consider a trace that is composed of M streams, with n_i references in each stream i. The address contained in the jth reference of the ith stream is $a_{i,j}$. We define the difference in addresses between "stream-adjacent" references as follows:

$$d_{i,j} = a_{i,j} - a_{i,j-1}$$

where $a_{i,0}$ is taken to be 0. Defining spatial locality as the mean logarithmic distance between stream-adjacent references, the spatial locality of an M-stream trace can be computed in units of bits as:

$$l_M = \frac{\sum_{i=1}^{M} \sum_{j=1}^{n_i} b_{i,j}}{\sum_{i=1}^{M} n_i}$$

where $b_{i,j}$ is the number of bits needed to represent $d_{i,j}$ in 2's-complement notation:

$$b_{i,j} = log_2(|d_{i,j} + \tfrac{1}{2}| + \tfrac{1}{2}) .$$

We can similarly define L_M as the spatial locality (average distance between stream-adjacent references) in units of bytes, given $B_{i,j}$, the number of bytes needed to represent $d_{i,j}$.

$$L_M = \frac{\sum_{i=1}^{M} \sum_{j=1}^{n_i} B_{i,j}}{\sum_{i=1}^{M} n_i}$$

L_M provides a simple information-theoretic measure of the information contained in a trace. The difference between L_M and the size of the addresses in a trace (e.g., 4 bytes) indicates the redundancy that may be removed by simply storing address differences rather than the addresses.

Measurements of spatial locality of memory references are tabulated below for traces of a variety of workloads taken from both CISC and RISC processors, using a simple M = 3 model wherein reads, writes, and instruction fetches are considered as separate streams. The table shows the fraction of all references whose $d_{i,j}$ requires one, two, or four bytes in sign-extended two's complement notation, along with the value of L_{rwf} in bytes and the type of traces measured.

Table 1: Spatial Locality (L_{rwf})

CPU	B ≤ 1	1 < B ≤ 2	2 < B ≤ 4	Locality	Workload
68020	0.770	0.050	0.180	1.59	Technical (user-only)
68020	0.794	0.051	0.155	1.52	Technical (user and O/S)
MIPS	0.867	0.062	0.070	1.27	SPEC92 (user-only)
MIPS	0.852	0.072	0.076	1.30	Utilities (user-only)
SPARC	0.842	0.059	0.099	1.36	SPEC92 (user-only)

With L_{rwf} roughly one-third the size of the addresses in the traces, it is clear that these traces could be compressed somewhat by simply storing address offsets instead of addresses. However, a stronger form of locality, namely sequentiality, can yield even more compression.

Sequentiality

Sequentiality is an extreme form of spatial locality in which references in a stream progress monotonically through contiguous memory locations. Observed within the CPU, a sequential stream has a uniform stride which is the size of the operands or instructions accessed; for example, $d_{i,j} = 4$ for a sequential stream of RISC instructions. At memory, the stride of a sequential reference stream will be the size of a cache line. (A more general definition of sequentiality would include any stream of references of uniform stride.)

On average, about 75% of $d_{i,j}$ in SPEC92 traces from RISC processors are exactly 4 bytes. As expected, over 90% of instructions are to sequential locations, with little variation over workloads or processor architectures. Data reference sequentiality, on the other hand, ranges from 0% to 68%, and is strongly dependent on both the workload and the processor.

Other Reference Stream Redundancies

Nearly half of all references in the traces we have examined are from the same stream and have the same offset as the immediately preceding reference. Such repetition in the reference stream carries very low information content, and could be represented quite efficiently by one instance of the repeated reference followed by a repetition count.

All of these characteristics of address traces — spatial locality, sequentiality, and repetition — constitute reference stream redundancies that may be used to compress those traces.

2.2 ASCII Encoding Redundancy

Use of an ASCII format for storing traces makes it easy to examine and edit traces while teaching and experimenting with trace-driven simulation. However, ASCII encoding is too inefficient for production use:

- An entire byte (8 bits) is used to store each hexadecimal digit, which contains at most 4 bits of information.
- The type of reference also occupies one byte, but conveys fewer than three bits of information.
- Using ASCII characters to separate fields and records is also highly redundant.

Twofold compression can usually be achieved simply by using binary rather than ASCII encoding.

3. PDATS Address Trace Compression

The PDATS (Packed Differential Address and Time Stamp) trace compression scheme was developed at the Parallel Architecture Research Lab (PARL) of NMSU, with a goal of reducing both trace file size and simulation time, without distorting the reference stream contained in the original trace. This can be viewed as a coding problem, in which bandwidth reduction is to be balanced against decoding complexity. For PDATS, a reasonable balance was found in employing the compression techniques suggested in the preceding section, with offset sizes quantized in byte units for processing efficiency.

Table 2 illustrates the steps used to produce a PDATS trace from the type and address fields of a dinero trace. The first column in Table 2 is the type of reference: 0 represents a data read, 1 a data write, and 2 an instruction fetch. This reference type field is also used in the PDATS file, but it occupies only 3 bits in each PDATS record.

The memory locations accessed are listed in the second column. PDATS converts these absolute addresses to address offsets, which are the differences between successive references *of the same type*, represented in two's complement using the minimum number of bytes required to hold the offset. The first address of each type is simply reproduced in the output file. When the original trace contains time stamps, non-negative offset encoding is employed to compress these.

Note that when successive references are of the same type and maintain a constant offset, a single instance of the type and offset specification, together with the number of times the first instance is repeated, suffices to describe the entire sequence of references.

Table 2: Example of Trace Processing

Input Trace		PDATS Trace		
Type	Address	Type	Offset	Rep
2	430d70	2	430d70	0
2	430d74	2	4	0
2	415130	2	–1bc44	0
0	1000acac	0	1000acac	0
2	415134	2	4	3
2	415138			
2	41513c			
2	415140			
2	430c20	2	1bae0	0
2	430c24	2	4	0
1	7fff00ac	1	7fff00ac	0
2	430c28	2	4	0
1	7fff00a8	1	-4	0
2	430c2c	2	4	0
1	7fff00a4	1	-4	0
2	430c30	2	4	0
1	7fff009c	1	-8	0
2	430c34	2	4	0
1	7fff00a0	1	4	0
2	430c38	2	4	0
1	7fff0098	1	-8	0
2	430c3c	2	4	7
2	430c40			
2	430c44			
2	430c48			
2	430c4c			
2	430c50			
2	430c54			
2	430c58			
1	7fff00b4	1	1c	0

3.1 PDATS Trace Encoding

A PDATS file is stored in a binary format consisting of a file header followed by variable-length records (from 1 to 8 bytes) in the following format:

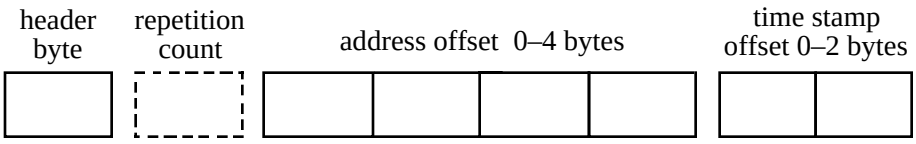

Figure 1: PDATS Record Structure

The structure of the header byte is shown in Figure 2, with the field encodings listed in Table 3.

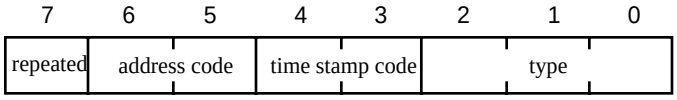

Figure 2: PDATS Header Byte Structure

The most straightforward approach to encoding address and time stamp offsets would simply use the minimum number of bytes necessary to hold each offset, with a tag in the header byte to indicate the number of bytes used. The coding efficiency of PDATS has been increased over this simple scheme by reserving a few codes in the header byte for specific high-probability offsets:

a. Approximately 85% of instruction fetches (from 32-bit microprocessors, both CISC and RISC) reference sequential memory words, as do many data loads and stores, making 4 byte offsets extremely common (about 75% of all references).

b. In scalar RISC microprocessors, the time between instruction references is frequently 1 clock cycle, making a time stamp offset of 1 very likely in RISC traces.

c. In Harvard-architecture microprocessors, data references can occur simultaneously with instruction references, leading to frequent occurrences of time stamp offsets of 0.

d. In superscalar microprocessors, time stamp offsets of 0 are also common, due to the issue of multiple instructions each clock.

When these cases occur, the corresponding address or time stamp offset is encoded entirely in the header byte, with the result that many references can be completely encoded in a 1-byte record.

When the repeat flag is set, the byte following the header byte specifies the number of times that this record is repeated (contiguously) in the original trace. This run-length coding can produce significant compression of instruction sequences, and requires minimal computation to regenerate the original reference stream.

Table 3: Header Byte Encoding

type	bit 2 - 0 (lsb)
0	user data read
1	user data write
2	user instruction fetch
3	escape record (unknown access type)
4	supervisor data read
5	supervisor data write
6	supervisor instruction fetch
7	interrupt acknowledge / co-processor / etc.
time stamp code	bit 4 - 3
0	time stamp increases by 0
1	time stamp increases by 1
2	time stamp offset is 2 - 255 (occupies 1 byte)
3	time stamp offset is 256 - 65536 (2 bytes)
address code	bit 6 - 5
0	address offset is exactly 4
1	address offset encoded in 1 byte (-128 to +127)
2	address offset encoded in 2 bytes
3	address offset encoded in 4 bytes
repeat	bit 7 (msb)
0	no repetition (repeat count byte absent)
1	repeat these offsets (repeat count present)

3.2 Trace Statistics Supporting PDATS Compression

A utility developed in the course of the PDATS project tabulates a number of interesting statistics from the contents of a PDATS trace, including the numbers of references of each type, and, for each type, a histogram of the number of bytes needed to store offsets, a histogram of run lengths,

and a histogram of time stamp offsets. An example set of statistics is shown below, which was collected for a 10 million reference trace of a SPARC processor executing the SPECint92 program espresso.bca.

```
Ref Type     Count
0 (Read)    1402312
1 (Write)    330960
2 (Fetch)   8266728
Total      10000000

              Address offset histogram (row = ref type):
          ---- Negative offset----  -------- Positive offset -------
Type      4 bytes  2 bytes  1 byte    +4    1 byte  2 bytes  4 bytes
0 (Read)   0.1434   0.2581  0.0794  0.0567  0.0608   0.2801   0.1214
1 (Write)  0.0936   0.0232  0.4506  0.1561  0.1474   0.0266   0.1025
2 (Fetch)  0.0031   0.0153  0.0271  0.8802  0.0600   0.0112   0.0031
Avg        0.0258   0.0497  0.0485  0.7408  0.0630   0.0494   0.0230

              Repeat counts histogram (column = ref type):
Repeats  0 (Read)  1 (Write)  2 (Fetch)
   0      1402312    330960    1564608
   1                            825929
   2                            528161
   3                            311586
   4                            219878
   5                             39960
   6                             19281
   7                             16673
   8                             19266
   9                              8156
  10                              2133
  11                               610
  12                             22309
  13                              1345
  14                               220
  15                               511
  16                                 7
  17                                 2
  18                               301
```

In traces produced by RISC processor simulators (or equivalent techniques), we have rarely observed any repeats in operand references. Only the AMD 29050 with its separate external instruction and data buses produced repeated operand accesses, and these were usually double-word reads, or multi-word writes for stack cache spills. For other RISC processors, only sequences of instructions exhibit repeated offsets, and these are always offsets of exactly 4 bytes (one RISC in-

struction). CISC traces, however, include both operand reference repeats and fetch repeats at a range of offsets. This holds for both Motorola 68020 bus traces (captured in hardware) and DEC VAX traces.

4. PDATS Compression Performance

The effectiveness of the PDATS technique in compressing various trace formats was evaluated by comparing trace file sizes before and after compression, and the time needed to read references from the original and compressed traces.

4.1 Traces Used in Evaluation

Table 4 summarizes the formats of the traces used in our evaluation. In addition to the popular dinero format, we used two in-house derivatives of dinero as well as the native (binary) file format produced by one of our hardware trace collection systems, a Tektronix DAS 9200. Dinero and its derivatives store trace references in variable-length ASCII character strings, with space characters embedded to separate fields and a newline character used to separate records. Our derivatives extend the basic dinero record format to include a time stamp (Green Stamp) or data needed to determine the original size of a trace that has been filtered (RATCHET I).

Table 4: Trace Formats Used in Evaluation

Trace Format	Source	Code	Record Size	Time Stamps
DAS	Tektronix Data Acquisition System	binary	20 bytes	yes
dinero	U.C. Berkeley (M. Hill)	ASCII	4-11	no
RATCHET I	NMSU (C. Schieber) for filtered traces	ASCII	9–16	no
Green Stamp	NMSU (M. Lumeyer)	ASCII	6–20	yes

The trace files used in the evaluation of PDATS are described in Table 5.

Table 5: Traces Used in Evaluation

Format/CPU	Trace	Program	References	User only	Timestamps	Filtered
DAS	gcc_all	gcc	131,009			
68020	spice_all	spice	98,301	n	y	n
(CISC)	gpssh_all	gpssh	98,301			
RATCHET I	gccbeg1	gcc	258,122			
68020	spmid1	spice	258,252	n	n	y
(CISC)	espbeg1	espresso	260,794			
Green Stamp 29050 (RISC)	espm2	espresso	448,730	y	y	n
dinero	cc1	gcc	1,000,002			
DLX	spice	spice	1,000,001	y	n	n
(RISC)	tex	LaTex	832,477			
dinero	lisp.000	lisp	291,390			
VAX	spic.000	spice	446,701	n	n	n
(CISC)	mul8.003	mul8	429,432			

For each trace format, traces were selected that represent a variety of memory reference behaviors. When possible, we included a trace of a C compiler (relatively irregular behavior), a Spice simulation (large data set with a mixture of looping and branching), and a third program that exhibits fairly repetitive behavior. The collection includes traces of both RISC and CISC processors, filtered and unfiltered traces, and traces that include supervisor and interrupt activity as well as user-only traces. (Many of these traces are available in the NMSU TraceBase: `ftp://tracebase.nmsu.edu`. See the list in the References section for more information.)

These relatively small traces (10^5 – 10^6 references each) were selected for breadth and efficiency in the initial evaluation of the PDATS technique. Because PDATS operates on very small neighborhoods of the trace file, the results obtained from these traces may be directly extrapolated to larger traces, as demonstrated at the end of this section.

4.2 Methodology

To individually evaluate the various components of the PDATS technique, the traces were recoded into four different formats:

- *Binary*: Each trace was first converted into an uncompressed binary form. Each reference was stored as 1 byte of type and 4 bytes of address. An additional 4 bytes of time stamp were included iff time stamps were present in the original file. Reduction of all of the input traces to this baseline format permits comparison of the information density in the original formats:

$$\text{Overhead} = 1 - \frac{\text{binary}}{\text{original}}$$

- *No Repeats*: The traces were compressed into packed differential form (Figure 1) without run-length (repeat) coding. The ratio of the resulting file size to the uncompressed binary file size yields a measure of the *locality* in the trace, including both spatial locality among the addresses and temporal locality among the time stamps:

$$\text{Locality} = 1 - \frac{\text{no rep}}{\text{binary}}$$

- *PDATS*: The traces were compressed into packed differential form, with run-length coding used whenever possible. This is the normal mode for PDATS. Any improvement in compression over the version without repeat coding is a result of *sequentiality* in the traces:

$$\text{Sequentiality} = 1 - \frac{\text{PDATS}}{\text{no rep}}$$

$$\text{PDATS Compression} = \frac{\text{original}}{\text{PDATS}}$$

- *LZ*: After compressing the trace using PDATS, Lempel-Ziv (LZ) compression was

applied to further compress the trace file. The substantial compression achieved in this step is a result of large scale patterns in the traces that are exposed by the PDATS algorithm (as discussed later).

The results of compressing the thirteen example traces are presented below, grouped by original trace format and CPU type. For each group, the sizes of the original and transformed files are shown graphically. A summary of the numerical results for all groups is listed in Table 6.

Table 6: PDATS Compression Results

Trace Type	Overhead	Locality	Sequentiality	PDATS Compression	PDATS + LZ Compression
DAS	55%	67%	3%	6.97	26.08
RATCHET I	67%	33%	1%	4.63	10.92
Green Stamp	37%	85%	26%	13.97	68.58
dinero (DLX)	47%	68%	12%	6.66	55.50
dinero (VAX)	52%	50%	9%	4.62	23.62

4.3 Compression of DAS Format Traces

The DAS format traces, gcc_all, spice_all, and gpssh_all, were captured in real-time from executions of the corresponding programs on an instrumented Sun–3/60 (68020 CISC CPU) running SunOS 4.0.3. These traces include user, supervisor, and interrupt activity. Every reference carries a time stamp. The DAS results are shown in Figure 3.

The original files are 2.23 times the size of the uncompressed binary files, for an "overhead" of 55%. The binary files, in turn, are 3.00 times the size of the files without repeat records, and the use of repeat records reduces the file sizes by another 3%. The average compression ratio using PDATS alone is nearly 7; with the addition of LZ compression, the ratio climbs to 26.

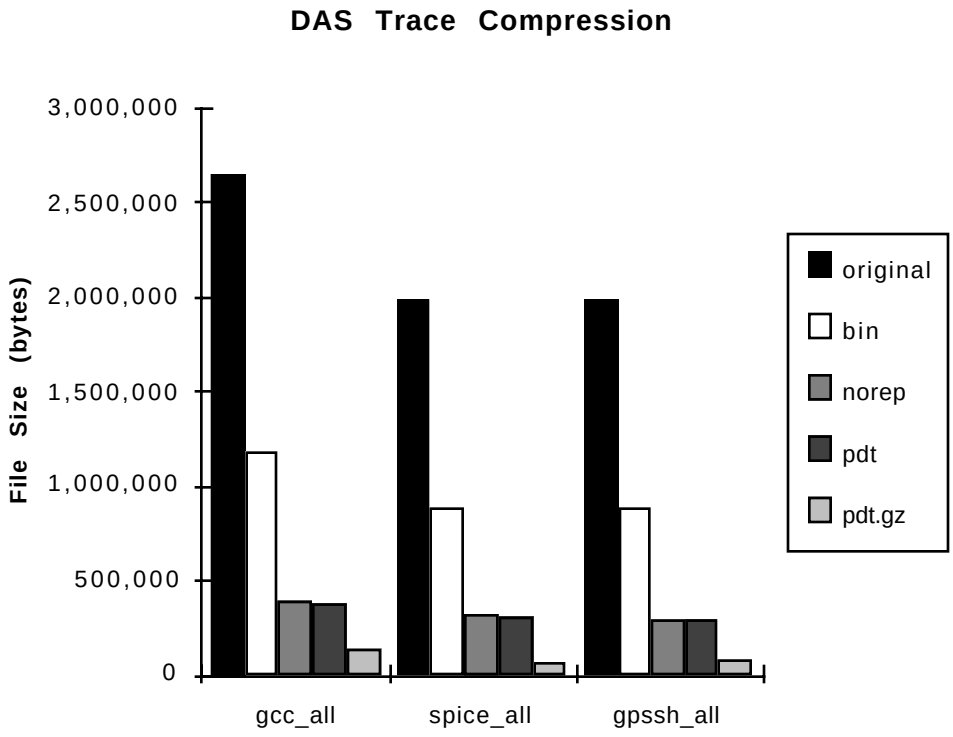

Figure 3: Compression of DAS Format Traces

4.4 Compression of RATCHET Format Traces

The RATCHET format traces, gccbeg1, spmid1, and espbeg1 were captured *and filtered* in real time from the instrumented Sun–3/60 described above, using the RATCHET I system [5]. Each trace comprises the contents of one of four DAS hardware buffers, converted into a format derived from dinero that includes reference type, address, transfer size, and a flag that is set (and forced through the filter) every 2^{16} actual (not filtered) references. Time stamps are not included.

The RATCHET results are shown in Figure 4. As expected, the trace filtering removed nearly all of the sequentiality from these traces, along with much of the spatial locality. PDATS alone achieved only fourfold compression. Even with Lempel-Ziv compression the ratio was only 11.

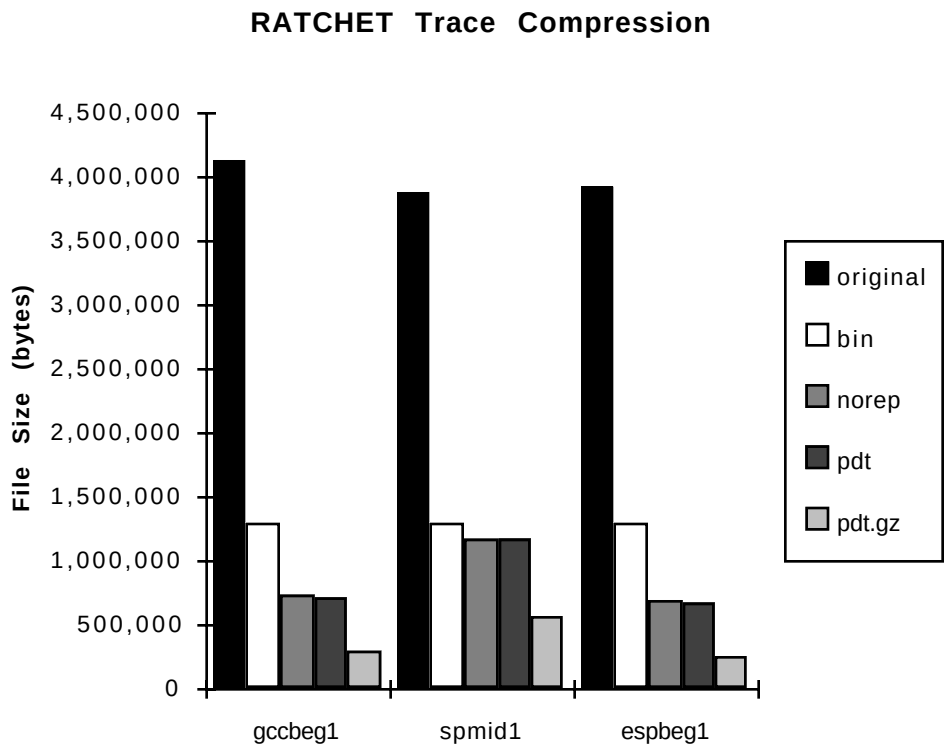

Figure 4: Compression of RATCHET Format Traces

4.5 Compression of Green Stamp Format Traces

The Green Stamp trace espm2 was generated from the SPEC89 program espresso, running on an Am29050 processor simulator. Supervisor and interrupt references are not included in the trace. Time stamps are included. The Green Stamp results are shown in Figure 5. This RISC processor is capable of extended instruction bursts and simultaneous instruction and data accesses (Harvard architecture), resulting in strong sequentiality and frequent time stamp differences of exactly 1. The latter two effects are encoded entirely within the PDATS header byte, requiring no offset bytes. As a result, PDATS was quite effective in compressing this trace, achieving a compression ratio of 15.42 alone, and over 75 when teamed with LZ.

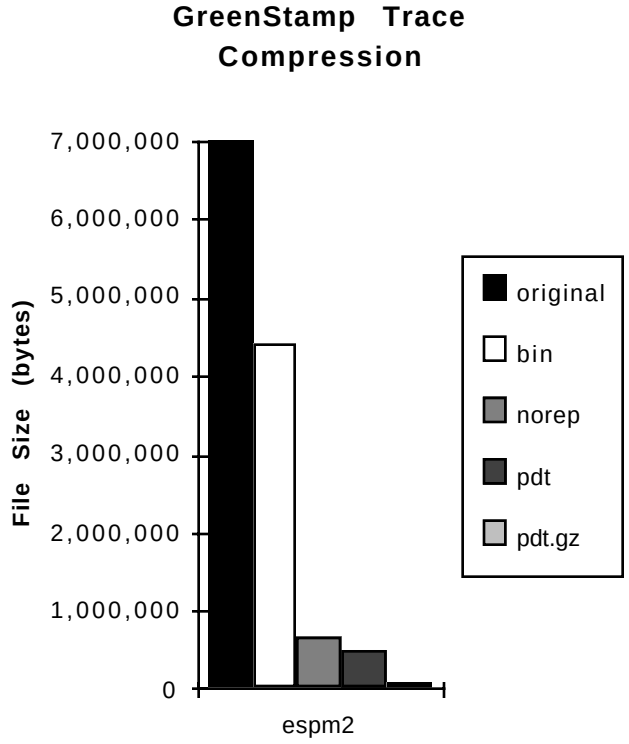

Figure 5: Compression of Green Stamp Format Traces

4.6 Compression of Dinero Format Traces

DLX is a “paper machine” used in a popular computer architecture text [10]. This processor incorporates characteristics of many of the first generation RISC processors. The cc1.din, spice.din, and tex.din traces were produced by a DLX simulator. These are in dinero format, and include only the standard dinero reference type and address fields in each record.

The results of compressing these traces using PDATS are shown in Figure 6. The locality and sequentiality are somewhat lower than for the other RISC trace (espm2 above), probably due to the absence of time stamps.

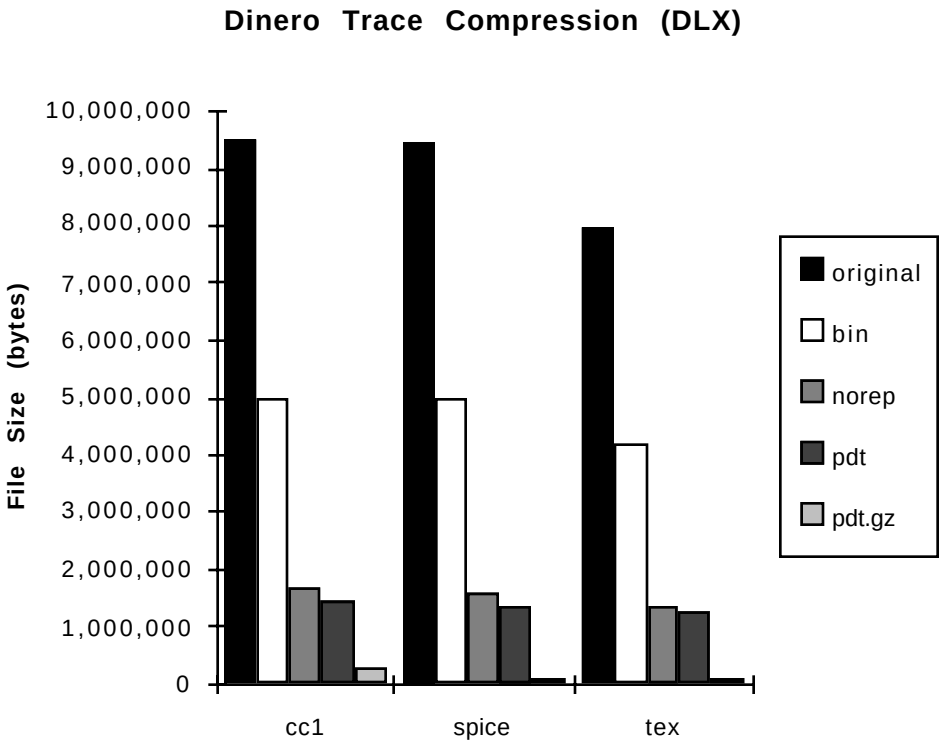

Figure 6: Compression of Dinero Traces from DLX

The lisp.000, spic.000, and mul8.003 traces are also in dinero format, but were obtained from a VAX using the ATUM technique [11]. Supervisor references are included, but time stamps are not. Results of compressing these traces are shown in Figure 7. The overhead is similar to the

DLX traces in dinero format, as would be expected. The lower locality and sequentiality are likely a result of the presence of both user and system references in these traces.

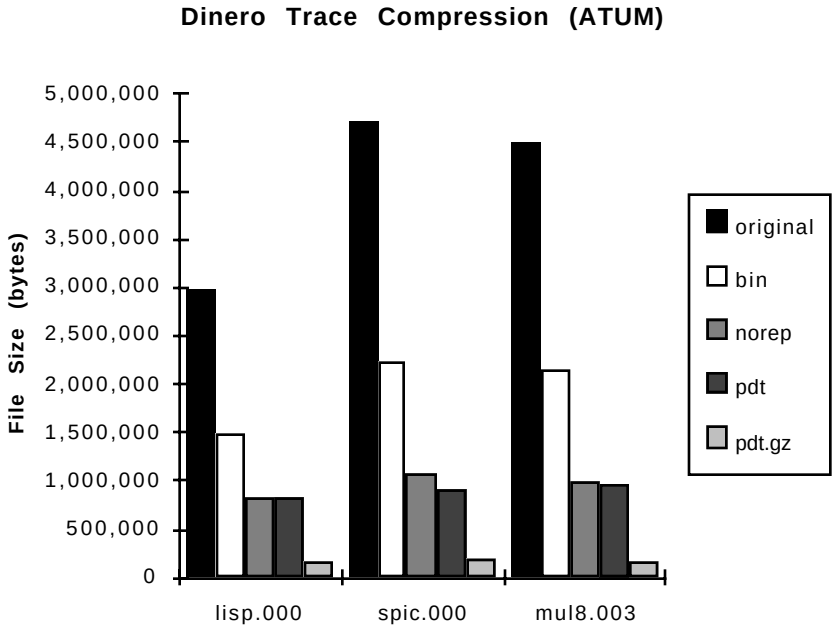

Figure 7: Compression of Dinero ATUM Traces from VAX

4.7 Compression of Longer Traces

Because PDATS operates on a scale smaller than the basic blocks within a trace, one would expect that the compression results obtained for the "toy" traces above will also apply to arbitrarily long "production" traces. As a check on this intuition, we compressed several traces of useful size, with the results shown in Table 7. The R2000 traces were collected by Nadeem Malik of IBM from execution of SPEC92 programs and the Unix utility awk. The SPARC traces were collected at NMSU using software instrumentation on SPARC 5 workstations.

When these results are compared with the compression ratios obtained for the shorter DLX traces, we see that the compression ratios are quite similar. We believe that the compression ratios obtained for most traces of interest will fall within the range of the results in Tables 6 and 7.

Table 7: Compression Results for Longer Traces

Trace	CPU	Dinero trace size (bytes)	References	PDATS Compression	PDATS + LZ Compression	PDATS + LZ ref's per byte
cexp	R2000	166,702,246	18,782,177	6.44	40.4	4.6
mdljd		738,116,943	84,233,871	8.49	88.1	10.1
awk		757,070,302	86,435,124	6.40	104.3	11.9
gcc	SPARC	814,267,402	100,000,000	5.99	37.3	4.6
espresso		74,471,978	10,000,000	6.02	53.9	7.2
li		79,752,827	10,000,000	4.94	25.1	3.1
ear		76,559,426	10,000,000	5.31	45.6	6.0
swm		79,325,250	10,000,000	6.08	252.4	31.8
tomcatv		73,206,405	10,000,000	5.83	43.7	6.0

4.8 Discussion of Compression Results

In general, we find that several characteristics of a trace will affect its compression under PDATS:

a. Traces from RISC processors have higher compression ratios than those from CISC processors. This is probably due to the greater proportion of instruction references in a RISC trace, as well as the longer basic blocks, both of which increase sequentiality.

b. Unfiltered traces have better compression ratios than filtered traces, because most of the spatial locality and sequentiality has been removed from filtered traces.

c. Single-thread traces have better compression ratios than “complete” traces that include multiple threads (in particular, supervisor or interrupt references). Unless context switches are especially frequent, multiple threads *per se* should not significantly affect the average size of address differences. The degradation observed in compression of complete traces is more likely due to irregular access patterns in supervisor and interrupt code.

d. Traces with time stamps have better compression ratios than those without time stamps, because time stamps usually increase by small amounts. Furthermore, when time stamp offsets are 0 or 1, PDATS achieves excellent compression by encoding the time stamp offset entirely within the header byte of the record.

In addition to eliminating much of the small-scale redundancy in traces, the PDATS coding scheme also exposes larger-scale patterns in traces that can be compressed very efficiently by Lempel-Ziv compression techniques, as discussed in the following section.

4.9 Exposure of Large-Scale Patterns

The family of compression techniques based upon the work by Ziv and Lempel [12] works by detecting common patterns in data streams and replacing them by small codewords. Such patterns can be quite large, but every instance to be replaced by a particular codeword must be identical. Thus, a dinero-format trace of a program loop that initializes an array, for example, would not be compressed particularly well by a Lempel-Ziv algorithm, because the operand addresses are different for each iteration.

However, after PDATS has converted the addresses in a trace to offsets, each iteration of a loop may contain identical offsets for both instruction and operand references. This would allow a Lempel-Ziv algorithm to replace an entire loop iteration with a single codeword, then pairs of iterations by another codeword, and so on until the entire array manipulation sequence is represented using only a few bytes in the compressed trace file. Because of the position independence of address offsets, a similar effect occurs whenever sequences of identical offsets occur at different absolute addresses in a trace.

The redundancy present in such large-scale patterns can be seen in the measurements of trace file sizes shown in Tables 8a and 8b. Six SPARC traces of SPEC92 programs were encoded in six different forms: dinero and PDATS in their native formats; compressed using Lempel-Ziv-Welch (*.Z); and compressed using Lempel-Ziv coding (*.gz). Although Lempel-Ziv coding can compress absolute address traces (dinero format) by an order of magnitude, an additional half-order-of-magnitude of compression can be obtained by using PDATS as the base trace file format, achieving a density of about six references per byte.

Table 8a: Compression Ratios from Dinero Format using L-Z Variants

Class	Program	din.Z	din.gz	pdt	pdt.Z	pdt.gz	ref/byte
SPEC92	gcc*	5.14	11.51	3.37	20.60	37.32	4.58
Integer	espresso.bca	7.20	19.28	6.02	24.49	53.91	7.24
(Cint92)	li	2.17	13.64	4.94	15.73	25.11	3.15
SPEC92	ear	7.41	13.35	5.31	30.77	45.60	5.96
Flt. Point	swm	11.56	85.82	6.08	94.43	252.36	31.81
(Cfp92)	tomcatv	9.37	24.19	5.83	36.55	43.74	5.97
	Cint92 avg	5.88	14.47	5.63	19.95	36.97	4.45
Averages	Cfp92 avg	9.29	30.26	5.73	47.36	79.54	8.18
	Overall avg	7.39	20.92	5.68	30.73	54.23	5.77

* All traces 10M references, except gcc which is 100M references

Table 8b: Compression Ratios from Timestamped Formats using L-Z Variants

Format	Program	.Z	.gz	pdt	pdt.Z	pdt.gz	ref/byte
GreenStamp	espresso.bca	2.90	3.43	13.97	34.94	68.58	4.79
	gcc	2.25	2.91	7.10	16.07	21.12	1.04
DAS	spice	2.84	4.26	6.71	23.55	33.52	1.66
	gpssh	2.41	3.11	7.08	20.71	28.71	1.43

The Mache compression scheme [8] used a subset of the techniques employed by PDATS, followed by Lempel-Ziv-Welch [13] compression. Mache difference files store offsets in a 2-byte header when they fit; otherwise a 4-byte offset is appended to the header. Repeat coding is not used. Before LZW compression, Mache is roughly half as effective as PDATS (i.e., Mache files are twice as large as pdt files). When LZW is applied to both, the differences in record size are largely hidden, but Mache files are still about 25% larger than pdt.Z files.

5. Evaluation of PDATS Speedup

The time to read traces compressed using PDATS was evaluated two ways. First, the time was measured to simply read traces in original, binary, and PDATS formats, including the conversion of each reference into its integer components. Steady improvements in CPU speeds increasingly render the time to read a trace from disk the dominant contribution to simulation time. Thus these measurements of access time speedup indicate the future value of PDATS in simulation speedup.

Currently, however, the time required to execute a significant number of instructions in processing each reference after it is read exceeds the time to read each reference. To estimate the current value of PDATS in speeding up cache simulations, the dinero III cache simulator was modified to read PDATS traces, and the running times of cache simulations using identical traces in dinero and PDATS formats were compared.

5.1 Access Time Measurements

The access times were measured by reading each trace from a SCSI RAID (level 5) attached to an IBM RS/6000 model 580 file server running AIX 3.2.5. The user plus system time spent in reading and converting each trace was measured using `/usr/bin/time`. These times were collected with the system in multiuser mode, but no other user jobs were running on the machine.

The results are summarized in Table 9, where the times for individual traces are summed within the groups. Times and speedups are listed for PDATS alone, PDATS with LZW and PDATS with LZ. The latter times reflect the time to uncompress the LZ or LZW encoding and pipe the result through the PDATS decoding program. It is interesting that LZ sometimes was faster than LZW. The uncompression times will, of course, decrease with improvements in CPU speed.

Table 9: Summary of Access Time Results

	Access Time (s)				Speedup		
Trace Format	din	pdt	pdt.Z	pdt.gz	pdt	pdt.Z	pdt.gz
DAS	0.73	0.40	1.10	1.13	1.81	0.66	0.64
RATCHET	3.97	1.07	3.47	4.03	3.72	1.14	0.98
Green Stamp	3.03	0.30	0.67	0.60	10.11	4.55	5.06
dinero (DLX)	11.63	1.83	4.53	3.70	6.35	2.57	3.14
dinero (VAX)	5.97	1.17	2.93	2.70	5.11	2.03	2.21

Access Time for DAS Format Traces

For DAS traces, the speedup achieved by PDATS over the original uncompressed file or the binary file is small, despite the significant compression achieved (a factor of 7 overall from the original

DAS traces). The principal reason for this is the variable-length records used by PDATS versus fixed-size records in both DAS and uncompressed binary formats. Nevertheless, even in this case PDATS achieved a net speedup in reading the traces because of the reduced file sizes. This speed-up could be improved by reading large, fixed-size blocks of the PDATS trace and parsing them in memory.

The overall data rate achieved by our test machine for this phase was 9.1 MB/s for the original DAS traces versus 2.4 MB/s for the PDATS traces. The 20 byte fixed-size fread() call used to read each record from the DAS file resulted in a sustained bandwidth close to the 10 MB/s peak bandwidth of the SCSI-II bus that connects the RAID to the server. However, the multiple small reads used to process each variable-length PDATS record substantially increased the I/O overhead.

Access Time for ASCII Format Traces

PDATS produces significantly larger speedups in reading traces that were originally in an ASCII format. This is due in large measure to the variable-length records in the ASCII format traces, as well as to the need to convert ASCII to integers. These result in roughly equivalent amounts of processing required to read either PDATS or ASCII formats, and speedups from using PDATS approximate the compression ratios.

For example, the set of RATCHET format traces takes nearly 4 times longer to read from the disk than the equivalent PDATS traces. We observed that the PDATS files take slightly longer to read than the uncompressed binary files, due to the absence of any significant spatial locality in these filtered traces. Again, the single fixed-size fread() used for the binary trace is sufficiently faster than the multiple reads for each PDATS record to overcome the much smaller PDATS files.

As an ASCII format with very good compression, the access time improvement for the Green Stamp trace was excellent, with a speedup of more than 10 from the original trace to the PDATS file. The dinero format traces exhibit qualitatively similar speedup results to the Green Stamp trace. However, the overall speedup is not as great (averaging 6.35 for DLX traces and 5.11 for VAX ATUM traces) because the compression ratios were similarly smaller.

Summary of Access Time Results

While less than twofold speedup was achieved relative to files already in binary form (DAS traces), the access time was improved by an order of magnitude for ASCII traces with time stamps, and by factors of roughly 4 to 6 for ASCII traces without time stamps. Use of fixed-length encoding rather than the variable-length scheme may incrementally improve the speed of reading traces, but the resulting increase in file size is unattractive.

For unfiltered ASCII format traces compressed using either of the Lempel-Ziv variants after PDATS, the time to uncompress and decode traces was always at least twice as fast as simply reading the trace in its ASCII form (Table 9), and the traces were 20 to 50 times smaller (Table 6).

5.2 Simulation Time Comparison

An alternative measure of speedup is the effect of the improved access times resulting from PDATS on the speed of a well-known trace consuming program. The `readfrominputstream`() function of the dineroIII cache simulator was modified to call a function that extracts references from a PDATS trace. This modified program is called dinerop.

The simulation times for the three DLX traces described previously are listed in Table 10, along with those for two 10 Mref traces from SPARC executions of the SPEC92 programs gcc and swm256. Because of the sophistication of the dinero simulator, a significant amount of processing is performed after each reference is read from the trace. This had the effect of diluting the access time speedups to a simulation speedup of 1.37 (i.e., dinerop runs 37% faster than dinero).

Currently, LZW compression (using the Unix compress command) is often used to reduce the size of dinero-format traces. The resulting file sizes are generally within half an order of magnitude of the equivalent PDATS files sizes (without LZW or LZ compression), which raises the question of how simulation times compare for pdt versus din.Z files. The measurements in Table 10 therefore also include the times to uncompress din.Z files using zcat, and pipe the resulting trace through dinero.

Table 10: Simulation Time Results

Trace	CPU	refs	Simulation Time (s)		
			din	din.Z	pdt
cc1	DLX	1 M	10.40	17.10	7.63
spice		1 M	10.30	16.67	7.57
tex		< 1 M	8.53	14.20	6.20
085.gcc	SPARC	10 M	101.90	154.78	73.80
078.swm		10 M	101.70	155.44	74.20

To summarize the simulation time comparisons, dinero simulations run about 37% faster when reading PDATS traces than when reading dinero traces, and more than twice as fast as simulations that uncompress LZW-compressed dinero traces “on the fly.”

6. PDI Compression of Instruction Traces

As noted in the Introduction, traces are used not only in simulations of memory hierarchies, but also in evaluations of CPU designs. For the latter class of applications, traces must contain both instructions and addresses. The PDI technique is an extension of PDATS for compressing traces that contain instruction words in addition to addresses. In PDI, addresses are compressed using a subset of the PDATS techniques described above, while the instruction words are compressed using a dictionary based approach, as discussed below.

6.1 Instruction Word Statistics

It is intuitive that some instructions are used more frequently than others. The resulting non-uniform probability distribution of particular instruction words constitutes redundancy that can be exploited to compress instruction traces. Although the technique is applicable to processors having fixed- or variable-length instructions, we here consider only fixed-length-instruction traces.

Analysis of instruction word usage in traces of MIPS processors executing SPEC92 benchmarks showed that not only are certain opcodes used more frequently than others, but certain complete instruction words (including the opcode, operand specifiers, mode bits and so on) occur very frequently in these traces. For the 20 SPEC92 traces tabulated below, the 256 most frequent in-

struction words accounted for 56% to 99.9% of the instructions executed, with a median hit ratio of 86% to the 256 most frequent instructions from each trace.

Table 11: Usage of 256 Most-Common Instructions

Trace	% hits	Int/FP
008.espresso	95.97	int
013.spice2g6	72.32	fp
015.doduc	68.08	fp
022.li	83.37	int
023.eqntott	80.78	int
026.compress	99.88	int
034.mdljdp2	78.60	fp
039.wave5	99.05	fp
047.tomcatv	99.56	fp
048.ora	97.10	fp
052.alvinn	75.65	fp
056.ear	88.38	fp
072.sc	85.77	int
077.mdljsp2	84.06	fp
078.swm256	99.47	fp
085.gcc	60.48	int
089.su2cor	91.53	fp
090.hydro2d	86.55	fp
093.nasa7	99.53	fp
094.fpppp	55.81	fp

It seems likely that this notion could be extended to include common instruction *sequences*, but this has not yet been pursued.

6.2 PDI File Format

A PDI file is a binary file containing a file header followed by a dictionary of the 256 most common instruction words found in that trace, followed by contiguous variable length records. The dictionary of high frequency instruction words is generated by analyzing the trace files and making a table of the 256 most common instruction words. The rest of the records range in size from 2 to 9 bytes in the following format:

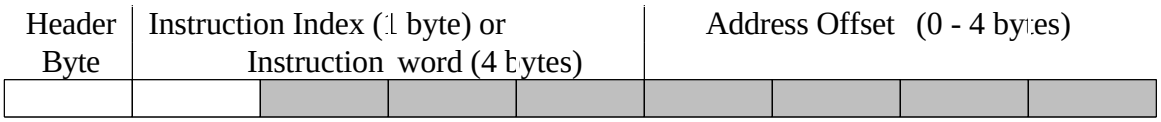

Figure 8: PDI Record Format

Table 12 illustrates the steps used to produce PDI trace from a dinero-style address + instruction trace. The first field (type) represents the operation being performed i.e. 0 = Data Read, 1 = Data Write, 2 = Instruction fetch. The second field contains the address of the location being accessed. The third field contains an instruction word being fetched (which is zero for data references).

Table 12: Example of PDI Trace Processing

Input Trace			**PDI Trace**		
Type	Address	Instr	Type	Offset	Instr
2	400190	8fa40000	2	400190	0
0	7ffebc64	0	0	7ffebc64	
2	400194	3c1c1001	2	4	1
2	400198	279cac50	2	4	3
2	40019c	27a50004	2	4	4
2	4001a0	af858d68	2	4	c
1	100039b8	0	1	100039b8	
2	4001a4	24a60004	2	4	9
2	4001a8	41080	2	4	6
2	4001ac	af808d6c	2	4	5
1	100039bc	0	1	4	
2	4001b0	00c23021	2	4	00c23021
2	4001b4	27bdffe8	2	4	2
2	4001b8	af868d60	2	4	d
1	100039b0	0	1	-c	
2	4001bc	afa00014	2	4	afa00014
1	7ffebc60	0	1	6ffe82b0	
2	4001c0	0c101570	2	4	c8
2	4001c4	af848d64	2	4	64
2	4055c0	27bdfe60	2	53fc	50
2	4055c4	afa401a0	2	4	3c

In the PDI trace, the first field represents the type of reference and is the same as the input type. The second field contains the address offset computed in normal PDATS fashion. The third field contains either an index into a dictionary of instruction words, or the instruction word itself if it was not one of the 256 most common instruction words for the particular trace.

6.3 Evaluation of PDI Compression

Clearly, the key to effective compression of instruction traces using a dictionary-based approach is the selection of an appropriate dictionary. Two techniques were investigated for producing such dictionaries. In the first, the trace to be compressed is read, and the instruction words found in it are histogrammed. The 256 most common instruction words are used as the dictionary for that trace. The PDI trace is then produced using a second pass through the original trace. This will yield the best possible compression, but at a cost of two passes through each trace to be compressed.

An alternative approach uses a "generic" dictionary that is selected for a particular processor/compiler combination after examination of a suitable collection of traces from that combination. This will usually yield somewhat less compression, but permits PDI trace generation in a single pass through the trace. This single-pass behavior is especially important when long traces are to be generated, since there may not be room to store the trace in an uncompressed format. A single-pass trace compressor can be included as part of a trace-producing pipeline, while a two-pass program cannot.

Results of applying the two PDI techniques are shown in Table 13. The traces used in this evaluation are 1 Mref traces of the SPEC92 programs collected from an R3000. The results in this table show the reduction in file size for each technique relative to the original dinero-style traces.

The *pdi (specific)* column lists the compression ratio of the PDI technique using a dictionary produced for each trace using the two-pass approach. The *pdi (generic)* columns reflects use of a dictionary that contains the 256 most common instructions from a composite histogram of all of the individual traces.

As expected, PDI uncovers patterns that can be used by Lempel-Ziv algorithms to achieve better compression than either PDI or Lempel-Ziv alone. As we saw previously for the PDATS technique, LZW achieves compression ratios for dinero-style traces comparable to PDI alone.

Table 13: Compression of SPEC92 Instruction Traces

Trace	din.Z	din.gz	pdi (specific)	pdi (generic)	pdi.gz (specific)	pdi.gz (generic)
Espresso	7.53	19.85	6.68	3.62	112.63	103.61
Spice	4.15	12.59	5.23	3.58	27.26	23.09
Doduc	5.31	10.80	5.11	3.64	56.42	41.52
Li	6.03	17.01	5.60	3.63	45.27	40.05
Eqntott	6.90	20.70	5.68	3.61	68.52	58.51
Compress	8.10	34.19	7.12	3.76	79.03	70.77
Mdljdp2	6.85	22.10	5.83	3.73	55.27	50.15
Wave5	9.39	86.54	7.01	3.67	437.24	324.23
Tomcatv	7.62	17.90	6.45	3.71	42.49	36.68
Ora	6.46	32.02	7.85	3.82	222.24	176.52
Alvinn	4.87	28.66	5.49	3.57	78.89	72.43
Ear	9.45	20.45	6.31	3.73	57.81	59.89
SC	6.55	19.42	7.79	4.90	78.97	68.37
Mdljsp2	7.32	25.28	5.84	3.64	49.80	45.28
Swm256	11.86	85.02	7.83	3.82	748.97	426.31
Gcc	4.32	7.71	4.86	3.57	44.52	38.82
Su2cor	6.78	18.93	6.96	3.86	190.61	150.93
Hydro2d	7.95	18.64	6.44	3.79	132.64	117.24
Nasa7	16.43	73.08	7.06	3.62	122.94	117.90
Fpppp	4.95	9.99	5.32	3.96	80.93	69.03
Median	6.88	20.15	6.38	3.69	78.93	68.70
Max	16.43	86.54	7.85	4.90	748.97	426.31
Min	4.15	7.71	4.86	3.57	27.26	23.09

The original dinero-style traces require about 16 bytes per reference. When PDI is followed by LZ, we achieve a coding density of about 4.5 references per byte for pdi (specific) and about 3.9 references per byte for pdi (generic). The latter density is only 33% less than for pdt.gz traces, which means that we can generate compressed instruction traces in a single pass that are only one third larger than address-only traces.

7. Conclusions and Future Work

Reduced storage requirements and shorter processing times are of obvious benefit to all users of traces. The PDATS technique described here provides both, while retaining all of the references of the original traces as well as any time stamps present.

Employed alone, PDATS achieves nearly order-of-magnitude improvements in storage space and access time, particularly for RISC processor traces. When Lempel-Ziv compression is applied after the trace-specific compression of PDATS, another half order of magnitude of compression can be obtained, although some of the access time improvement may be lost.

As a result of these findings, PDATS is in use as the standard trace format for a data base of traces established at NMSU for the use of researchers and educators worldwide. (Our goal is to provide easy access to a large body of useful traces so that researchers will be able to reproduce and extend each others' work, a capability long enjoyed by researchers in the physical sciences.)

Future investigations will pursue techniques for improved compression. We have observed that offsets between operand references often require four bytes. Our hypothesis is that this occurs when accesses to stack and heap areas occur in alternation. This suggests that the use of additional streams in analyzing a trace (e.g., one for stack references and another for the heap) would reduce the average size of operand reference offsets. Other patterns in reference streams will be studied to identify additional opportunities to remove trace redundancy.

An ongoing application of the PDATS compression scheme is hardware implementations to compress traces "on the fly" during real-time trace acquisition. When combined with some degree of trace filtering, it may be possible to reduce trace data bandwidth to that sustainable by a disk array, allowing the capture of extremely long traces in real time that include user, system, and interrupt activity. We are pursuing this idea in a project called RATCHET. RATCHET III produced PDATS traces in real time from a 20 processor Sequent Symmetry backplane. Similarly, RATCHET IV captures and compresses traces in real time from an IBM F-50 multiprocessor.

References

1. M.D. Hill, DineroIII Documentation, Unpublished Unix-style Man Page, University of California, Berkeley, October 1985.

2. A.J. Smith, "Two Methods for the Efficient Analysis of Memory Address Trace Data," *IEEE Transactions on Software Engineering*, vol. SE-3, no. 1, 1977.

3. T.R. Puzak, "Analysis of Cache Replacement Algorithms," Ph.D. Dissertation, University of Massachusetts, 1985.

4. A. Agarwal and M. Huffman, "Blocking: Exploiting Spatial Locality for Trace Compaction," *Proceedings, ACM SIGMETRICS 1990*.

5. E.E. Johnson and C.D. Schieber, "RATCHET: Real-Time Address Trace Compression Hardware for Extended Traces," *Performance Evaluation Review*, vol. 21 nos. 3 and 4, pp. 22-32, April 1994.

6. S. Das and E.E. Johnson, "Accuracy of Filtered Traces," *Proceedings of IEEE International Phoenix Conference on Computers and Communications*, pp. 82–86, April 1995.

7. J.W.C. Fu and J.H. Patel, "Trace Driven Simulation using Sampled Traces," *Proc. Twenty-seventh Annual Hawaii International Conf. on System Sciences*, pp. 211-220, 1994.

8. A.D. Samples, "Mache: No-Loss Trace Compaction," *Proceedings, ACM SIGMETRICS 1989*: pp. 89-97.

9. E. E. Johnson and J. Ha, "PDATS: Lossless Address Trace Compression For Reducing File Size and Access Time," *Proc. IEEE International Phoenix Conference on Computers and Communications*, pp. 213-219, May 1994.

10. J.L. Hennessy and D.A. Patterson, *Computer Architecture – A Quantitative Approach*. San Mateo, CA, Morgan Kaufmann, 1990.

11. A. Agarwal, R.L. Sites, and M. Horowitz, "ATUM: A New Technique for Capturing Address Traces Using Microcode," *Proceedings, 13th Annual International Symposium on Computer Architecture*: pp. 119-127, 1986.

12. J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory*, vol IT-23, No. 3, May 1987.

13. T.A. Welch, "A Technique for High-Performance Data Compression," *IEEE Computer*, vol. 17, no. 6, pp. 8-19, 1984.

Traces

The following traces used in this paper are available in the NMSU TraceBase:

ftp://tracebase.nmsu.edu/pub/

tracebase3/

sparc/

- 008.espresso.bca.10m.pdt.gz
- 022.li.10m.pdt.gz
- 023.eqntott.10m.pdt.gz
- 026.compress.10m.pdt.gz
- 047.tomcatv.10m.pdt.gz
- 056.ear.10m.pdt.gz
- 072.sc.10m.pdt.gz
- 078.swm.10m.pdt.gz
- 085.gcc.100m.pdt.gz
- 085.gcc.10m.pdt.gz

tracebase4/

68020_complete/

- gcc_all.pdt.gz
- gpssh_all.pdt.gz
- spice_all.pdt.gz

29k/

- espm2.pdt.gz

r2000/SPEC92/

- spec034.mdjld.pdt.gz
- spec085.cexp.pdt.gz

r2000/utilities/

- awk.pdt.gz

r3000/pdt/

- 008.espresso.pdt.Z
- 013.spice2g6.pdt.Z
- 015.doduc.pdt.Z
- 022.li.pdt.Z
- 023.eqntott.pdt.Z
- 026.compress.pdt.Z
- 034.mdljdp2.pdt.Z
- 039.wave5.pdt.Z
- 047.tomcatv.pdt.Z
- 048.ora.pdt.Z
- 052.alvinn.pdt.Z
- 056.ear.pdt.Z
- 072.sc.pdt.Z
- 077.mdljsp2.pdt.Z
- 078.swm256.pdt.Z
- 085.gcc.pdt.Z
- 089.su2cor.pdt.Z
- 090.hydro2d.pdt.Z
- 093.nasa7.pdt.Z
- 094.fpppp.pdt.Z

Affiliation of Authors

Eric E. Johnson, Jiheng Ha, and M. Baqar Zaidi

Klipsch School of Electrical and Computer Engineering

New Mexico State University

ejohnson@nmsu.edu

Acknowledgment of Financial Support

This work was supported in part by the U.S. Army Research Office under grant DAAH04–93–G–0229.

Previous Publication

Preliminary evaluation of the PDATS technique was published in the proceedings of the 1994 International Phoenix Conference on Computers and Communications, IEEE. (See reference [9]).

Contact Information

Contact Author: Eric E. Johnson

Klipsch School of Electrical and Computer Engineering

New Mexico State University

Las Cruces, NM 88003

Internet: ejohnson@nmsu.edu

Phone: (505) 646-4739

Fax: (505) 646-1435