PDATS II: Improved Compression of Address Traces

Eric E. Johnson
Klipsch School of Electrical and Computer Engineering
New Mexico State University
ejohnson@nmsu.edu

Abstract

The tremendous storage space required for a useful data base of memory reference traces has prompted a search for trace compaction techniques. PDATS is the standard trace format used in the NMSU TraceBase, a widely used archive of memory and instruction traces. The PDATS family of trace compression techniques achieves trace coding densities of about six references per byte — with no loss of reference type or address information — by using differential run-length encoding. This paper proposes an improvement on the PDATS scheme that doubles the typical compression ratio without losing information.

1. Introduction

Processor architects have long used trace-driven simulation of new CPU designs to identify performance problems early in the design process. Likewise, memory system designers have used traces of memory references to tune hierarchies for optimum performance on targeted workloads. Often, these traces must be quite large: using cache simulation as an example, the evaluation of megabyte-size caches requires traces with hundreds of millions to billions of references. If the memory hierarchy design is to perform well over a wide range of applications, it must be tuned during the design phase using a similarly wide variety of traces. Thus, a trace library for production use must contain many billions of references. The disk space needed to store such a trace library equals the product of the number of references and the number of bytes needed to store each reference. Even with today's low cost per gigabyte of disk space, we would like to reduce the latter term far below the 8–10 bytes/reference typical of the well-known dinero trace format [1] as long as none of the information used by trace-driven simulations is lost in the process.

The two lossless address trace compression schemes reported in the literature are Mache [2] and PDATS [3]. The two are similar in concept, but PDATS achieves somewhat better compression and is in widespread use as the standard format for address traces in the NMSU TraceBase [4]. We begin our discussion with a review of the PDATS scheme.

2. PDATS address trace compression

Lossless trace compression must remove only redundancy from the trace; it cannot discard useful information. As an introduction to the redundancy inherent in the easy-to-use dinero format, consider the following trace segment. A trace in the dinero format contains ASCII characters that specify the reference type (0 = read, 1 = write, and 2 = instruction fetch) followed by the reference address in hexadecimal. A blank is used as a field separator, and a newline is used as the record separator:

Type	Address
2	430d70
2	430d74
2	415130
0	1000acac
2	415134
2	415138

It is clear, even from this short segment of a trace, that most of the characters used in dinero instruction records are redundant, since instructions are nearly always executed sequentially. Although not evident in this trace segment, data references exhibit strong locality as well. Also, representing numbers in ASCII rather than binary roughly doubles the size of each record [3].

The PDATS (Packed Differential Address and Time Stamp) format retains the reference type (read, write, or

0-7803-5258-0/99 $10.00 © 1999 IEEE

fetch), the address, and the time stamp (if present) of each trace record, but does a reasonable job of eliminating the redundancy that that is due to spatial locality and sequentiality. PDATS converts the absolute addresses in the trace to address offsets, which are the differences between successive references *of the same type*. These address offsets are represented in two's complement using the minimum number of bytes required to hold the offset. When successive references are of the same type and maintain a constant offset, a single instance of the type and offset record, together with the number of times the pattern is repeated, suffices to describe the entire sequence of references.

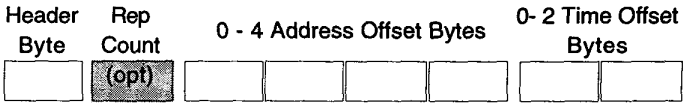

Figure 1: PDATS record structure

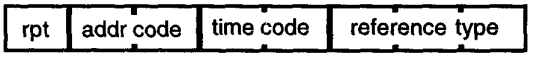

Figure 2: PDATS header byte structure

2.1 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; see Figure 1). The structure of the header byte is shown in Figure 2, with the field encodings listed in Table 1 (with the exception of the time code, which is not addressed in this paper).

When the repeat flag is set, the byte following the header byte holds 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 little computation to regenerate the original references.

Table 1: PDATS header byte encoding

repeat	bit 7 (msb)
0	no repetition (repeat count byte absent)
1	repeat these offsets (repeat count present)
addr code	bit 6 - 5
0	address offset is exactly 4 (default offset)
1	address offset encoded in 1 byte (-128 .. +127)
2	address offset encoded in 2 bytes
3	address offset encoded in 4 bytes
type	bit 2 - 0 (lsb)
0	user data read
1	user data write
2	user instruction fetch
others	(supervisory or system references)

2.2 PDATS compression performance

To establish a baseline for the effectiveness of the PDATS technique, six SPARC traces of SPEC92 programs were encoded in six different forms: dinero and PDATS in their native formats; compressed using Lempel-Ziv-Welch [5] (*.Z); and compressed using Lempel-Ziv [6] 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 was obtained by using PDATS as the base trace file format, achieving a density of about six references per byte when PDATS is followed by Lempel-Ziv compression, as seen in Table 2.

$$\frac{ref}{byte} = \frac{references}{final\ file\ size}$$

Table 2: PDATS compression ratios for 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

3. Opportunities for further compression

Although PDATS achieves impressive reductions in trace file size, examination of the resulting compressed traces reveals a number of inefficiencies that offer opportunities for additional compression. These result from common patterns of program behavior, which are illustrated in the following discussion using traces of the SPEC92 programs `gcc` and `swm256` on R3000 and SPARC processors.

3.1 Jump/post-jump encoding

Jumps, i.e., non-sequential instruction fetches, are often followed by several sequential instruction fetches. PDATS encodes this common pattern using one record to specify the jump, followed by another record (often including a repeat count) that describes the sequential references. The first record requires more than one byte because the offset of the jump is by definition different from the default instruction stride that can be encoded within the PDATS header byte. The following sequential instructions have default stride, but a repeat count adds another byte to the second record, for a total of at least four bytes for this common sequence.

We would like to encode all such instruction sequences using a single record. To determine the feasibility of this, consider the statistics on run lengths shown in Figure 3 for a non-sequential instruction fetch followed by sequential fetches with no intervening data references.

It is clear from the figure that few jump-initiated sequences are longer than 15 contiguous instructions. Thus, if we can use at least four bits in the header byte to encode run length, we can usually eliminate the second record used by PDATS to represent a jump-initiated sequence.

3.2 Data-instruction clusters

One reason for the plateau in Figure 3 past instruction sequences of just a few fetches is that the instruction fetches of a basic block are usually broken into numerous short sequences by intervening data references. We refer to a data reference followed by one or more instruction fetches as a "data-instruction cluster."

PDATS encodes each data-instruction cluster using at least two records. The first describes the data reference, and includes at least a header byte; extra bytes for data offset are also present in most cases. The following instruction fetches can be described in two bytes iff they are the sequential continuation of the instruction sequence that began before the data reference.

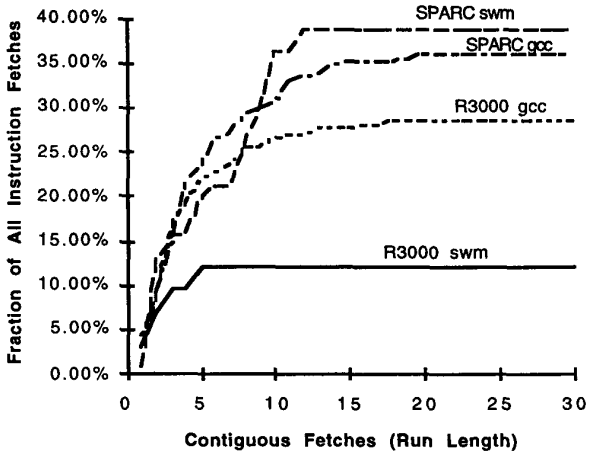

Figure 3: Cumulative fraction of instruction fetches in jump-initiated sequences

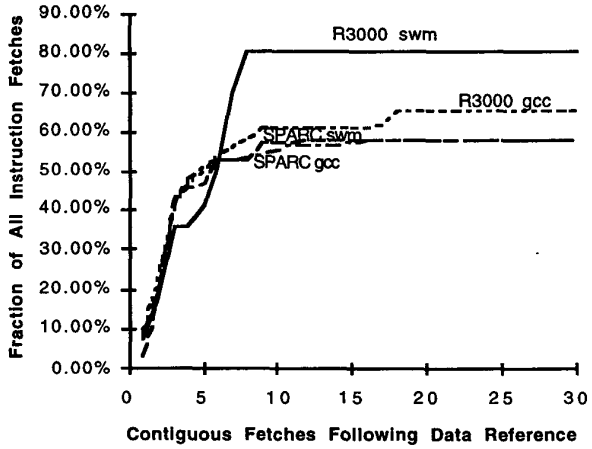

Figure 4: Cumulative fraction of instruction fetches in data-instruction clusters

These data-instruction clusters are the dominant feature of most traces, representing an even larger fraction of all references than the jump-initiated sequences just described (see Figure 4). We would therefore like to encode each data-instruction cluster in a single byte if possible. From the figure it appears that we can capture most data-instruction cluster lengths in three bits.

3.3 Common Data Offsets

The common offset +4 encoded in the PDATS header is quite powerful in reducing the encoded size of instruction fetches; however, there are other very frequent *data* offsets that currently require an offset byte, as shown in Figure 5. If the most common of these could be captured in the header byte, many data offset bytes could be eliminated.

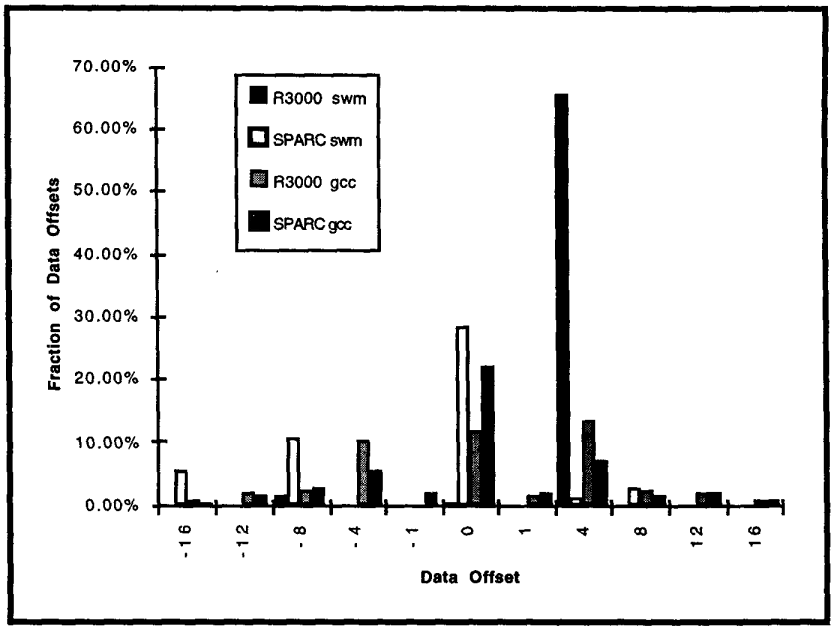

Figure 5: Common data offsets

3.4 Operand Zones

Data references in many programs are clustered in a few widely separated regions of the address space, a result of complier/loader conventions that establish distinct locales for stack, heap, and so on. When references alternate among these areas, the resulting offsets are quite large (often requiring the full 4-byte encoding), even though the offsets between references *within* each area exhibit substantial locality.

3.5 Read-Write Locality

Separating the read and write reference streams is not very effective in reducing offsets. When a program is writing to a particular memory locale, it is also very likely to be reading from that locale. When read and write streams are considered separately for computing offsets, the result is usually *two* large offsets whenever the locale changes.

3.6 Default Instruction Stride

Although roughly 90% of instruction offsets are exactly +4 bytes (for the RISC processors in common use today), the other 10% have offsets that must be efficiently encoded in an integral number of bytes if reading and decoding a trace file is to be fast. We can gain some additional compression by storing such instruction offsets in units of the default instruction stride (i.e., dividing instruction offsets by 4 bytes). This will sometimes permit an offset to fit into one or two bytes that would otherwise require two or four bytes for 2's-complement representation.

4. PDATS II Trace Compression

The research just described led to an improved PDATS — called PDATS II — that uses all of the characteristics of program behavior listed in the previous section to more effectively compress address traces. As noted later, however, it wasn't always possible to allocate as many bits in the header byte as the analysis above indicates as desirable.

A PDATS II file is stored in a binary format similar to that of PDATS, consisting of a file header followed by variable-length records (from 1 to 5 bytes) as shown in Figure 6. (A 64-bit-address version supports up to 8 offset bytes.) The timestamp and supervisor/system reference features of PDATS were eliminated to free some bits in the PDATS II header byte.

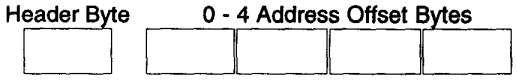

Figure 6: PDATS II record structure

Because of the different statistics of offsets and run lengths for data *versus* instruction references, the header byte encoding differs depending on whether a jump-initiated sequence or a data-instruction cluster is described in a PDATS II record.

4.1 PDATS II instruction records

About 90% of instruction offsets in RISC traces equal the instruction size, so it is clear that one instruction offset code should represent this size. From Figure 3, we see

that at least four bits should be allocated in the instruction-record header byte to count sequential instructions following the first instruction fetch reported by this record. With one bit needed to distinguish instruction from data headers, we could either use three bits for the instruction offset code and four for the sequential instruction count or two bits for offset codes and five for the count.

Better compression results if we use 5 bits for the instruction count, and use two bits to select an instruction offset from exactly +4, a 1-byte range, a 2-byte range, or a 4-byte range. Thus, we arrive at the instruction-record header byte format shown in Figure 7:

Figure 7: PDATS II instruction header byte

4.2 PDATS II data records

Because data references lack the strong sequentiality of instruction fetches, we need to encode several common data offsets in the header byte. Allowing three codes for the 1-, 2-, and 4-byte ranges, we can include 5 common offsets in a 3-bit coding scheme, or 13 in a 4-bit scheme. It is clear from Figure 5 that little would be gained by including more than the 5 most-common offsets, which are 0, ±4, and ±8 bytes, so the 3-bit scheme was chosen.

With one bit necessary to differentiate data from instruction headers, and another required to distinguish reads from writes, a 3-bit offset code field leaves only 3 bits for the fields that count instructions following the data reference and select among active data regions in the working set. Both approaches — counting instructions and distinguishing data "zones" — yield substantial compression, so neither should be entirely eliminated. Analysis of program behavior indicates that a compromise of two data zones and instruction runs of up to 3 yields the best compression. Thus the data header byte in PDATS II is constructed as shown in Figure 8.

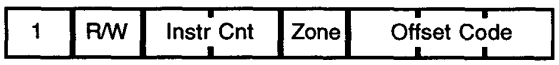

Figure 8: PDATS II data header byte

4.3 An example

An example of compressing a dinero trace using both PDATS and PDATS II is shown in Table 3. (This trace segment was taken from the beginning of the `tex` trace from a "DLX" processor [7].) The number of bytes needed to store each record is listed in the two rightmost columns of the table. Several of the points made earlier about opportunities for improvement in the original PDATS format are illustrated here:

1. The first two references illustrate the jump/post-jump situation: where PDATS requires one record to report the jump followed by another to describe the subsequent sequential instructions, PDATS II encodes the jump and the sequential instruction in a single record.
2. The third reference, a backward jump, requires a four-byte offset in PDATS, but only a two-byte offset in PDATS II due to its scaling of instruction offsets by the default instruction stride. Thus `-1bc44` is stored as `-6f11`, or `90ef`.
3. The fourth through eighth references illustrate both the power and the weakness of the PDATS II data header: it is able to encode the data read and the three following sequential instruction fetches in a single record, but an additional one-byte record is required when the instruction burst following a data reference exceeds three instructions.
4. However, the string of alternating writes and instruction fetches that begins with the twelfth reference provides a dramatic illustration of the improvements in PDATS II: where PDATS requires three bytes to store ascending instruction fetches alternating with descending writes, PDATS II can represent each pair with a single byte. This is due to its encoding of data strides of -4 in the header, along with its ability to store short instruction runs in the data header.

Note that both data zones have been set by the end of this example: the read placed the zone 0 base address in what may be the heap area, while the sequence of writes (which appear to be stack pushes), set the zone 1 base address much higher in memory. The advantages of the two-zone scheme in PDATS II are not apparent in this short trace segment, but become important very quickly as reads and writes in the two active data zones alternate between the high and low addresses. PDATS frequently requires large offsets to encode these data references, while PDATS II is able to encode many of them using the small-offset codes available within its data header.

Table 3: Example of Trace Processing

	Input Trace		PDATS Trace			PDATS II Trace				Bytes Needed	
#	Type	Address	Type	Offset	Rep	Type	Zone	Offset*	Count	PDATS	PDATS II
1	2	430d70	2	430d70	0	2		10c35c	1	5	5
2	2	430d74	2	4	0					1	
3	2	415130	2	-1bc44	0	2		-6f11	0	5	3
4	0	1000acac	0	1000acac	0	0	0	1000acac	3	5	5
5	2	415134	2	4	3					2	
6	2	415138									
7	2	41513c									
8	2	415140				2		1	0		1
9	2	430c20	2	1bae0	0	2		6eb8	1	5	3
10	2	430c24	2	4	0					1	
11	1	7fff00ac	1	7fff00ac	0	1	1	7fff00ac	1	5	5
12	2	430c28	2	4	0					1	
13	1	7fff00a8	1	-4	0	1	1	-4	1	2	1
14	2	430c2c	2	4	0					1	
15	1	7fff00a4	1	-4	0	1	1	-4	1	2	1
16	2	430c30	2	4	0					1	
17	1	7fff009c	1	-8	0	1	1	-8	1	2	1
18	2	430c34	2	4	0					1	
19	1	7fff00a0	1	4	0	1	1	4	1	2	1
20	2	430c38	2	4	0					1	
21	1	7fff0098	1	-8	0	1	1	-8	3	2	1
22	2	430c3c	2	4	7					2	
23	2	430c40									
24	2	430c44									
25	2	430c48				2		1	4		1
26	2	430c4c									
27	2	430c50									
28	2	430c54									
29	2	430c58									

* Offset field indicates true offset for data records, offset/4 for instruction records

5. PDATS II compression performance

The effectiveness of the PDATS II technique in compressing address traces was evaluated by comparing trace file sizes in dinero, PDATS, and PDATS II formats, with and without further Lempel-Ziv compression.

The results are presented in Table 4. The compression ratio is given for each trace for each output format, along with average compression ratios for the integer and floating point programs and the overall average compression ratios. The average number of references represented by each byte in the output files is also listed. These measurements are from 10 Mref traces from a SPARC processor. Other RISC processors produce similar results.

PDATS II generally compressed these traces twice as effectively as PDATS, and achieved an overall average of 13.4 references per byte when combined with Lempel-Ziv post-compression, or 1.33 references per byte without. (The additional compression achieved by the Lempel-Ziv algorithm results from recognizing and compressing large-scale patterns such as loop bodies.)

It is interesting to observe that the compression ratios for floating point traces differed little from those for integer traces for both PDATS and PDATS II acting alone. The small-scale patterns that these compression techniques recognize are about the same in both types of program. However, the large-scale behavior is much more redundant in the floating-point traces, as seen in the dramatic differences in compression ratios between integer and floating point when either LZ or LZW is applied afterward.

The superior compression performance achieved by PDATS II makes it quite attractive for the capture, storage and even generation of traces. We are developing a backward-compatible decompression filter that recognizes and transparently decompresses both PDATS and PDATS II traces, and hope that it will be quite useful.

References

1. Hill, M.D., dineroIII documentation, unpublished Unix-style Man Page, University of California, Berkeley, October 1985.
2. Samples, A.D., "Mache: No-Loss Trace Compaction," *Proceedings, ACM SIGMETRICS 1989*: 89-97.
3. Johnson, E. E. and Ha, J., "PDATS: Lossless Address Trace Compression For Reducing File Size and Access Time," *Proc. IEEE International Phoenix Conference on Computers and Communications*, 213-219, May 1994.
4. The TraceBase at NMSU can be found at <http://tracebase.nmsu.edu>.
5. Welch, T.A., "A Technique for High-Performance Data Compression," *IEEE Computer*, **17** (6): 8-19, 1984.
6. Ziv, J. and Lempel, A., "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory*, vol IT-23, No. 3, May 1987.
7. Hennessy, J.L. and D.A. Patterson, *Computer Architecture – A Quantitative Approach*, Morgan Kaufmann, San Mateo, CA, 1990.

Table 4: PDATS II Compression Performance

	Dinero		PDATS			PDATS II		
Program	din.Z	din.gz	pdt	pdt.Z	pdt.gz	pdt	pdt.Z	pdt.gz
gcc	4.91	16.89	6.16	24.36	62.59	10.99	37.24	89.25
espresso.bca	7.20	19.28	6.02	24.49	53.91	10.64	32.48	78.07
li	5.50	13.64	4.94	15.73	25.11	10.61	46.87	100.06
tomcatv	9.37	24.19	5.83	36.55	43.74	8.13	34.54	52.13
ear	7.41	13.35	5.31	30.77	45.60	10.71	82.19	311.87
swm	11.56	85.82	6.08	94.43	252.34	10.83	182.02	408.60
Cint92 avg	5.87	16.60	5.71	21.53	47.20	10.75	38.86	89.13
Cfp92 avg	9.45	41.12	5.74	53.92	113.89	9.89	99.58	257.53
Overall avg	7.66	28.86	5.72	37.72	80.55	10.32	69.22	173.33
Avg refs/byte	0.91	2.49	0.74	3.60	6.33	1.33	6.25	13.40