Arithmetic Program Paths

Manos Renieris
er@cs.brown.edu

Shashank Ramaprasad
shashank@cs.brown.edu

Steven P. Reiss
spr@cs.brown.edu

Computer Science Dept.
Brown University
Providence, RI 02912, USA

ABSTRACT

We present Arithmetic Program Paths, a novel, efficient way to compress program control-flow traces that reduces program bit traces to less than a fifth of their original size while being fast and memory efficient. In addition, our method supports online, selective tracing and compression of individual conditionals, trading off memory usage and compression rate. We achieve these properties by recording only the directions taken by conditional statements during program execution, and using arithmetic coding for compression. We provide the arithmetic coder with a probability distribution for each conditional that we obtain using branch prediction techniques. We implemented the technique and experimented on several SPEC 2000 programs. Our method matches the compression rate of state-of-the-art tools while being an order of magnitude faster.

Categories and Subject Descriptors

D.2.5.s [**Software Engineering**]: Testing and Debugging-Tracing; E.4.a [**Data**]: Coding and Information TheoryData compaction and compression

General Terms

Experimentation, Measurement, Performance

Keywords

Bit-tracing, Arithmetic Coding, Branch Prediction

1. INTRODUCTION

Program control-flow traces can be valuable aids to reverse engineering [2, 25], debugging [24], and program comprehension and are essential for dynamic slicing [1, 8] and instruction scheduling. However, at the speed of modern processors, control-flow traces of even a few seconds of execution can be too large to store on disks, transfer on networks, let alone to manipulate and analyze. A number of researchers have therefore addressed the problem of compressing program control-flow traces.

A successful attack on the problem will have:

- a high compression rate,
- time efficiency,
- space efficiency,
- the ability to produce compressed traces as the subject program is running,
- the ability to record only selected statements, for example, only specific functions,
- the ability to trade-off memory and time for improved compression rate.

The state-of-the-art solutions are based on Sequitur, a compression algorithm that discovers structure in sequences and encodes that structure as a context-free grammar that, when fully expanded, produces a single string: the original sequence. Sequitur is a general-purpose algorithm that can be applied on any kind of sequence and works hard to discover its structure.

The key insight in this paper is that much of the structure of program traces is either available statically or can be quickly discovered during an execution, (as witnessed by feedback directed optimization), so that the compression algorithm need not discover the structure as it runs. The technical challenges are representing this structure and using it effectively for compression; we meet the first challenge by using ideas from branch prediction and the second by using arithmetic coding. The main advantage of our approach, which we call Arithmetic Program Paths (APP) is increased speed, but also flexibility and a fixed memory footprint.

Arithmetic Program Paths are *arithmetically encoded bit traces*. A bit trace [3, 7, 9] is a representation of the control-flow component of a program that includes a single bit for each evaluation of a conditional during the run. The bit is "1" if the conditional evaluated to `true` and the `then` clause was taken, and "0" in the contrary case. Given the program, a bit trace contains enough information to reconstruct the whole control-flow trace, that is, the exact sequence of executed basic blocks. Arithmetic coding is a highly effective compression technique that can compress even binary alphabets. It requires, at each encoding/decoding step, a probability distribution over the possible input symbols, and its efficacy strongly depends on the accuracy of this distribution. We construct such a distribution using ideas from branch prediction.

We implemented a source-level instrumenter and collected arithmetic program paths from a number of programs in the SPEC 2000 suite[1]. In most cases, our method matches Sequitur's compression ratios while being an order of magnitude faster.

The rest of this paper is structured as follows: Section 2 discusses program control-flow tracing. Section 3 discusses arithmetic coding, and Section 4 discusses branch prediction. Section 5 describes APP, Section 6 describes our implementation, and Section 7 describes our experimental findings. Section 8 presents our conclusions and future work.

2. PROGRAM TRACING

Program-trace compression has attracted much interest over the past 15 years. Researchers have focused on compressing control-flow traces or full traces, i.e. traces that include memory references and values. Full traces necessarily include a control-flow trace, possibly encoded separately. Reported compression rates are not always directly comparable, because they depend strongly on the program, the execution's length, and the representation of the raw (uncompressed) trace.

The simplest way of tracing a program execution is to record at each conditional statement the value of the condition. This produces a sequence of zeros and ones that can be used, post-mortem, to reconstruct the executed control-flow.

Larus [14] introduced Whole Program Paths (WPP), based on the profiling work with Ball [3]. The main idea of [3] is to instrument a carefully selected number of edges of the control flow graph, so as to allow efficient counting of the executions of acyclic intraprocedural paths. Larus [14] used this technique to collect and output a sequence of acyclic intraprocedural path identifiers. He then used Sequitur [21] to compress the sequence into a *whole program path.*

Sequitur produces a context-free grammar whose language contains a single string, the original uncompressed string. When the original string contains a repeated substring, the repetitions are encoded using the nonterminals created to encode the first occurrence of the substring. The process is built upon two invariants: that no consecutive pair of symbols appears more than once in the grammar, and that no nonterminal appears less than twice. Sequitur's structure discovery creates the opportunity to compress repeated substrings at various levels of granularity, and Sequitur compresses structured constructs extremely well: Larus reports compression rates between 7 and 400. On the other hand, Sequitur maintains its invariants by keeping each consecutive pair of symbols in the grammar in hashtable which needs to be accessed for every symbol in the original string. In addition, Sequitur must keep significant part of the grammar in memory at all times, as any part of the grammar may require modification when new rules are built.

Zhang and Gupta [33] pointed out that whole program paths obtained in [14] are accompanied with increased difficulty in accessing path traces of subsets of information. They present an alternate *compaction* technique that removes redundant path traces and maintains subtraces for individual routines. Their technique hinges on two passes over the trace, since they need to construct a dynamic control-flow graph. They report compaction rates[2] between 7 and 64.

Pleszkun [23] developed a two-pass algorithm for compressing full traces. The first pass over the trace constructs the dynamic control-flow graph of the program. The second pass generates a Huffman-encoded description of the path taken through the dynamic control-flow graph during the particular execution.

Burtscher and Jeeradit [5] and Burtscher [4] present *vpc2* and *vpc3*, respectively, that compress a data-address stream using a dynamic Huffman compression scheme. A key idea in these tools is the use of a *value predictor* that predicts for every instruction the values involved, including the addresses. When the prediction is correct, it suffices to encode a single bit indicating this. If not, it is necessary to encode first a bit indicating that the encoder was incorrect and then the actual address. With a highly accurate predictor a single bit suffices most of the time, and therefore this scheme compresses well. Burtscher's techniques use Sequitur to encode the control flow of the program.

3. ARITHMETIC CODING

In this section, we discuss the cornerstone of our technique, arithmetic coding. Arithmetic coding allows encoding of binary alphabets, as long as we provide a probability distribution. To encode control-flow traces as arithmetic program paths, all we need to encode is the direction of each conditional branch because, given the program, the current branch, and its outcome, we can find which conditional branch we will encounter next. Therefore, if we have a distribution for the direction of each conditional branch, we can use arithmetic coding to compress control-flow traces. In the next section, we discuss a method, similar to branch predictors that can provide this distribution.

3.1 Coding

Most compression schemes are built around a coding component, that simply maps input symbols to output symbols. Morse coding maps letters of the alphabet to sequences of dots and dashes. As Morse knew, it is beneficial to map frequent inputs to short outputs: this reduces the size of the message. Huffman coding [10] automates the assignment of short output binary strings to frequent input symbols. It is based on building a binary tree whose leaves are the possible input symbols. The tree is balanced with respect to the probabilities of individual input symbols. The code for each input symbol is a description of the path from the root to the leaf that corresponds to the symbol.

Under specific circumstances, Huffman coding is optimal, in the sense that it achieves Shannon's bound. In his seminal 1948 paper [27], Shannon proved that no coding scheme can do better, on average, than encode a symbol i in $-\log_2 p_i$ bits, where p_i is the probability of encountering symbol i in the input. If S is the set of symbols that can appear in the input, an ideal coding scheme that achieves this bound will allocate, on average, $H = \sum_{\forall x_i \in S} p_i \log(1/p_i)$ bits to an input symbol. This value, known as the *entropy* of the source, bounds the efficiency of any coding scheme. Huffman coding achieves this bound when all the input symbol probabilities are integral powers of $1/2$.

Shannon's theoretical bound proved difficult to meet in practice in the general case when input symbols have arbitrary probabilities, in part because meeting it requires representing input symbols in a non-integral number of bits if their probability is not a power of $1/2$. For example, say we have two symbols A and B with probabilities $P(A) = 1/1000$ and $P(B) = 999/1000$, respectively. The entropy of this source is only about 0.11, implying that it is theoretically possible to compress a sequence of these symbols to about 11% of its original size. However, the shortest possible coding that uses an integral number of output bits per input symbol is (up to symmetry) $A \mapsto 0$, $B \mapsto 1$. Such a coding scheme achieves no compression. In fact, a scheme (such as Huffman's) that uses an integral number of output bits per input symbol can never compress if the input alphabet is binary.

Arithmetic coding can encode input symbols in fractional numbers of bits, asymptotically achieving Shannon's bound. It was first developed in the 1960s and 1970s [22, 26], although it was present as a research problem and a core idea in Shannon's original paper. Arithmetic coding maintains a subinterval of $[0, 1)$ to encode the input string. The encoding process starts with the unit interval $[0, 1)$. At each step, it associates each potential input symbol with a subinterval of the current interval, where the width of the subinterval associated with symbol i is of length proportional to the probability of i. For example, in the first step of encoding a sequence for our example $P(A) = 1/1000$ and $P(B) = 999/1000$ source, the $[0, 1)$ interval is split into the interval $[0, 1/1000)$ representing A and the interval $[1/1000, 1)$ representing B. Then the actual symbol to be encoded is examined, and the subinterval associated with it becomes the current interval. If we encounter A, then the current interval becomes $[0, 1/1000)$, which is to be subdivided in $[0, 1/1000^2)$ and $[1/1000^2, 1/1000)$. The algorithm thus deals successively with smaller, nested subintervals. The encoder can return any real number within the final interval; this number is called the *codeword*.

The decoder receives the codeword from the encoder and essentially undoes the encoding operation. Like the encoder, it begins with the interval $[0, 1)$. In every iteration, the decoder associates each potential input symbol with a subinterval of the current interval. Then the decoder finds the subinterval within which the codeword lies, outputs the the symbol associated with the subinterval, and makes this subinterval the current interval.

The decoding process as described above can proceed indefinitely, since the final interval can be indefinitely subdivided and still contain the code string. To avoid this, the encoder can explicitly pass the length of the sequence to the decoder, so that the decoder stops after decoding that many symbols. Another solution is to augment the input alphabet with a special EOF symbol that signals the end of the input.

This description of arithmetic coding cannot be implemented directly: the encoder writes the output codeword only after it has processed all the input, the decoder must keep the whole codeword in memory, and the codeword itself needs to be computed with infinite precision. These obstacles were surmounted with techniques invented in the 1970s: the codeword can be written and read incrementally, and the arithmetic can be done with finite precision. These advances allow implementations in which the whole state of the arithmetic decoder/coder is the bounds of the interval.

We should also note that the process can encode/decode even binary alphabets and can encode perfectly any probability distribution that does not require more than the precision available for the codeword computation. Details can be found in [12, 18, 31].

3.2 Modeling

So far, we have not discussed where the coder and decoder obtain the probability distributions for the next possible symbol. An attractive characteristic of arithmetic coding is that the probability distribution is entirely decoupled from the coding process. The only requirement on the distribution is that the coder and decoder agree on it. In fact, the probability distribution can change as the process evolves.

This has three significant effects. First, the probability distribution need not pertain to a constant alphabet; if we are compressing sequences of letters from the Latin alphabet and the coder and decoder both know that the next symbol will be a vowel, not a consonant, they can both switch to the probability distribution and the interval subdivision for vowels. Therefore, using arithmetic coding, we can encode diverse alphabets in the same stream.

Second, the probabilities of symbols can be discovered as the process unfolds. The coder and decoder can start with a uniform probability distribution over the alphabet, count how many times a symbol has appeared so far, and use such counts (normalized) as the probability distribution for the next step. This process is called *adaptive modeling*. Adaptive modeling is as efficient as any two-pass based modeling. In other words, if we compress with a two-pass scheme, in which the first pass computes (and stores) the exact counts and the second pass uses those exact counts to compute the probabilities and encode the string, the two-pass scheme ends up using as much space as an adaptive scheme [6].

Third, the probability distribution can change depending on the context, that is on a number of recently encoded/decoded symbols. Since both encoder and decoder have seen the same symbols before attempting to encode/decode a new symbol, they can use a probability distribution that is conditional on the recently seen symbols. Which and how many recently seen symbols to use as context is the core of the modeling problem. For arithmetic program paths, we use ideas from branch prediction to address this context problem.

4. BRANCH PREDICTION

Arithmetic compression requires us to give the coder a probability distribution for the next symbol. For arithmetic program paths, where symbols correspond to the direction taken by conditionals or (in lower-level terms) conditional branches, we can obtain a probability distribution with a modified branch predictor, in which the typical two-bit counter is replaced by a pair of integer counters. In this section, we describe branch predictors in general; our modification is described in section 5.

Dynamic branch predictors are present in all modern processors and are indispensable in superscalar processors [28]. Their purpose is to reduce the cost of branch instructions. Modern processors can execute multiple instructions at one time. The efficacy of multiple instruction schemes depends, however, on the ability of the processor to execute the correct sequence of instructions; i.e., the processor must start working on instructions that will actually be executed. The

```
if (x == 0) {
   y = z;
}
if (x >= 0) {
   print(a)
}
```

Figure 1: A program fragment with correlated branches. If the first conditional is true, so will be the second.

processor can always determine with certainty which instructions are to be executed next except at conditional and computed jumps. In those situations fetching the wrong instruction makes the processor backtrack, wasting time.

For the common case of conditional jumps, the solution lies with branch predictors, which provide a hint as to whether the jump is likely to be taken or not. Fetching can start at the address the branch predictor indicates will be executed next. If the branch predictor is correct often enough, the correct instruction is fetched and the processor will wastes less time.

Branch predictors can be static or dynamic. Static branch predictors do not exploit the execution history of the conditional jump; instead, they use various heuristics to determine which direction the jump is likely to take at compile or instruction decoding time. Dynamic branch predictors base their hint on the recent history of the conditional that is about to execute and also on the history of other recently executed conditionals.

Modern dynamic predictors have two parts [32]: an indexing part and a prediction machine part. The indexing part chooses one among many prediction machines; the prediction machine then provides the actual prediction. A prediction machine can be as simple as a single bit that remembers the last outcome of this branch and uses it as the prediction, or a two-bit counter that predicts the direction taken most often during the last three executions of this particular conditional. Smith and Nair [29, 19] both found that counters with more bits do not improve performance significantly, and therefore two-bit counters are generally used.

A basic indexing part can use simply (part of) the address of the conditional jump. In addition, indexing parts can use "local" context, i.e. bits from recent executions of the same conditional, and "global" context, i.e. bits from recently executed conditionals, whichever those may be. The use of local context is effective when the conditional follows a pattern; the use of the global context is effective in cases where the conditionals are related, as for example in Figure 1. Since hardware branch-predictor implementations are frugal in the number of gates they employ, all these kinds of bits are usually multiplexed, for example [16] by xor-ing bits from the local and global contexts. Much research has investigated optimal combinations of global and local contexts, with often impressive prediction accuracy reaching 99%.

Of particular interest here is a form of *alloyed* branch prediction [15]. Alloyed branch prediction concatenates bits from the conditional's address, a number of global context bits, and a number of local context bits to produce the index. This form of indexing is relatively simple to implement in software, since it avoids the associative tables common in hardware branch predictors [28].

5. ARITHMETIC PROGRAM PATHS

The key technical insight in this paper is that we can use branch-prediction techniques to feed an arithmetic coder in order to get good compression with a constant memory footprint. This section discusses how this is achieved, while the next section details our implementation and presents experimental results.

5.1 The Process

Arithmetic Program Paths (APP) has two sides: recording (together with compression) and playback (together with decompression). The interesting moments during execution are the control transfers with multiple possible targets, which happen at conditional statements (if statements, switch statements, and loops), and also indirect jumps (function pointer calls, virtual function calls, exceptions, and C setjmp/longjmp calls). We first describe how to handle conditionals, the simple case.

APP uses adaptive modeling; Thus the distributions for individual conditionals are not static during execution but evolve as the execution proceeds. Therefore, during recording, at every conditional the system must:

1. Evaluate the condition.

2. Tell the arithmetic coder the current probability distribution for the current conditional and the value of the condition, storing the compressed data.

3. Update the probability distribution for the current branch, taking into account the condition value.

4. Resume regular execution and proceed to the next conditional.

During playback, the system must

5. Tell the arithmetic decoder the distribution for the current conditional (which must match the distribution used during encoding) to decode one boolean value.

6. Update the probability distribution for the current conditional, taking into account the just decoded value.

7. Use the decoded value to replay the original system's control flow until the next conditional.

Steps 1 and 4 are part of the original program. Steps 2 and 5 can be handled using readily available libraries for arithmetic coding; as we noted in section 3.1 the implementation of arithmetic coding maintains one essential piece of data from one encoding/decoding step to the next, i.e. the coding interval, represented as two integers.

Step 7 requires that the program's control-flow graph is available in some form during replay.

This can be achieved either by keeping the program, by keeping its control-flow "shell" (see Figure 2), or by explicitly emitting the control-flow graph during the recording phase. In the first case, any processing of the encoded data must be interleaved with the program control-flow shell; in the second case, a more direct approach can be used.

```
while (i != EOF) {
   read(x);
   a = x + 3;
   if (a > 10) {
      print(a)
   }
}
```

BECOMES

```
while (read_bit()) {
   if (read_bit()) {
   }
}
```

Figure 2: A program fragment and its control flow skeleton

5.2 Modeling: From Branch Prediction to Probabilities

The most interesting steps in section 5.1 are steps 3 and 6, where the probability distribution for a conditional is updated. The design we use is based on the alloyed branch predictor [15] described in section 4.

For the indexing part, alloyed predictors use some bits from the conditional jump instruction's address, some local history bits, and some global history bits. Since we implement Arithmetic Program Paths in software, we can enumerate the conditionals in the code and use each conditional's number instead of the program counter part. Additionally, we can use l bits of local context and g bits of global context for each conditional.

We replace each of the two-bit counters in the prediction machine part with two integer counters. Two-bit counters cannot be used directly to provide the probability distribution necessary for arithmetic coding of control-flow traces. For example, when the counter is at an extreme (0 or 3) a zero probability is implied for the not-predicted direction. Should an error in prediction happen in this state, it would have to be encoded with an infinite number of bits. With two integer counters, we can initialize both of them to 1 (to avoid the zero-probability problem), increment one of them each time the branch is taken, increment the other one each time the branch is not taken, and use each counter value divided by the counter sum as the probability of each direction.

This scheme gives us a probability distribution for the outcome of a conditional that is conditioned on the outcomes of the last l occurrences of this conditional and the outcomes of the g most recently executed conditionals. To represent all these distributions, if $|\mathsf{if}|$ is the number of if statements in the program, we need a table of $|\mathsf{if}| \cdot 2^{(l+g)} \cdot 2$ counters. If the number of conditional statement is prohibitively large we can multiplex conditionals by using only some of the lower-order bits of conditional number to address the table.

5.3 Handling Indirect Jumps

There are two schemes for handling indirect jumps:

- The *symmetric* scheme [17] performs a program analysis to find the possible targets at each indirect jump, and then converts the jumps to a set of cascading if statements that check the pointer for equality with each possible target and then perform the jump explicitly. If a total order between the possible targets is available, we can utilize a binary search for the current

```
while (i != EOF) {
   // loop body
}
```

BECOMES

```
while (1) {
   while_0_continue: if (! (i != EOF)) {
      goto while_0_break;
   } // loop body
} while_0_break:
```

Figure 3: Sample CIL loop transformation

target. We call this the symmetric scheme because, once we perform this transformation on the program, recording and playback can be instrumented as in the absence of function-pointer calls.

- The *asymmetric* scheme instruments each indirect jump target to output an identifier in the encoded stream, while in the playback the branch reads the identifier and performs a direct jump. An asymmetric scheme can also be a starting point for handling `setjmp` and `longjmp`.

6. IMPLEMENTATION

This section describes the implementations of Arithmetic Program Paths and Whole Program Paths that we compare in section 7.

6.1 Instrumentation

The first step in collecting arithmetic program paths is instrumenting the subject program. For each subject program, two programs are created: a recorder program, which stores the control-flow trace into a file, and a playback program, which follows the original program's control flow, by reading a trace while calling client functions at every conditional, function call and return.

We first describe the recorder part. Our instrumentation system is built on top of CIL [20], an Ocaml source-to-source transformation library for the C language. We first apply two of CIL's own transformations to the subject program. The first transforms all loops into simple conditional statements, as in Figure 3. The second transforms all binary boolean operators to sequences of nested `if` statements, as in Figure 4. Both transformations insert `goto` statements as necessary. CIL also converts all `switch` statements to sequences of `if`s and `goto`s.

Our current implementation handles function-pointer calls with the symmetric scheme of section 5.3, using Steensgaard's analysis[3] [30] as the pointer analysis.

Our instrumentation technique, although relatively simple and effective, has the drawbacks of any source-to-source transformation. First of all, it cannot handle precompiled libraries. Second, since we record only the direction of conditionals, should any instrumented code be called from uninstrumented code, we lose and are unable to reclaim the correct sequence of conditionals. This can happen when the main program passes callback functions to library functions, for example `qsort`.

---

[3]Implemented by John Kodumal as part of CIL.

```
if (i == '#' || i == '@') {
   // conditional body
}
```

BECOMES

```
if (i == '#') {
   // conditional body
} else {
   if (i == '@') {
      // conditional body
   }
}
```

Figure 4: Sample CIL boolean transformation

After these transformations, all control flow information is contained in `if` statements. We then instrument the `then` branch of every `if` statement to pass a 1 to the arithmetic coder and the `else` branch to pass a 0. After the bit is encoded, we update the global and local contexts and also the probability distribution for the current `if`.

The values of l and g are command-line parameters. We seed each one of the contexts with the value of all true and we seed all counters with equal values.

We used the implementation of arithmetic coding described in [18], which is available from Alistair Moffat's web page[4], along with its efficient special-case routine for coding binary alphabets. We implemented one significant optimization, by splitting the encoding of zeros and ones into two functions, to let the compiler better optimize the library code. With binary alphabets, the cost of implementing an EOF symbol (see section 3.1) is not reasonable, so we relied on the following trick: since the playback program is following the control flow of the recorder program, it exits after reading precisely as many bits as were written by the recorder. Therefore, there is no need to encode explicitly the length of the trace. On the other hand, if unexpected exits are possible (for example, the program can abort), it is easy to encode the trace length with a sufficiently long integer.

The model is represented as a three-dimensional array of counter pairs: the first dimension is the `if` statement number, the second the bits of local context, and the last the bits of global context. To avoid excessive memory overheads we use only the ten low bits of the number of the conditional. The arithmetic coding library we use requires each counter pair to be represented as three integers (allowing a more efficient implementation of division). The memory overhead for any program is thus $|if| \cdot 2^{(l+g)} \cdot 3 \cdot 4$ bytes, on machines with 4-byte integers. For example, with 1024 `if` statements, $l = g = 7$, the memory overhead is 192 megabytes.

For the playback part, we implemented an instance of Larus's Abstract Execution [13]. We use CIL to extract all computation from the program: assignments, function calls to external functions, etc., are all removed. What remains is the control-flow shell of the program (see section 5.1): conditionals, including loops, and calls to functions with a defined body. Conditional expressions are replaced with calls to an external library, which will decode the required bit from a previously recorded trace. Updating the local and global contexts, as well as updating the counters, is done as in the compression stage.

---

[4] http://www.cs.mu.oz.au/~alistair/arith_coder/

Program	Time (sec)	Bit trace (MB)	Acyclic Path (MB)	Size Ratio	Edges/ Path
gzip	37	788	4782	6.0	2.6
vpr	28.8	372	1678	4.5	3.5
mcf	27.3	187	1557	8.3	1.9
crafty	19.9	356	1938	5.4	2.9
parser	9.1	136	1219	8.9	1.7
vortex	9.9	233	1082	4.6	3.4
bzip2	64.7	932	5681	6.0	2.6
twolf	13.6	175	1435	8.1	1.9
art	21.1	88	913	10.3	1.5
equake	43.1	104	1432	13.7	1.1

Table 1: Subject programs, trace sizes, and relation between bit traces and acyclic paths

6.2 Whole Program Paths

For our experiments with Whole Program Paths, we implemented a simple off-line version. We use 16-bit program path identifiers, since some programs have more than 255 distinct paths. For the data collection, we used a hash table based technique to convert bit traces into acyclic paths. For compression, we used the advanced Sequitur implementation available from Nevill-Manning[5]. This implementation of Sequitur uses a hash table to store the consecutive pairs of symbols in the grammar, outputs a symbol as soon as possible, and compresses the resulting grammar arithmetically. The hash table has about 62 million entries and occupies about 237 megabytes. Sequitur's dynamic memory consumption is larger, since it must also store the grammar.

7. EXPERIMENTS

7.1 Subjects, Setup, and Basic Statistics

We experimented with a set of programs in the SPEC 2000 suite, shown in table 1. The limitations of the symmetric scheme in handling callbacks and `setjmp/longjmp` pairs (see section 6) inhibited the inclusion of some SPEC 2000 programs. In the case of `parser`, which uses a `qsort` function we overcame this difficulty by adding Doug McIlroy's `qsort` implementation[6].

We used version 3.3.4 of gcc, the GNU C compiler[7] as the back-end compiler. CIL is stricter than gcc (especially with respect to agreement between the types of formal and actual arguments), which necessitated small changes to the SPEC programs. We used the *train* datasets that come with the SPEC benchmarks. All timings were collected on a machine with AMD Athlon64 3200+ with 1.5 GB of memory, running 32-bit Linux. All tracing programs were writing on the local disk. For programs in which the benchmark contains more than one run, we report total time and sum of trace sizes.

Table 1 shows the running time for each subject when compiled without optimization and run on the *train* dataset. When instrumented, they produced bit traces of the sizes shown (in megabytes) in column 3. Column 4 gives the size of the corresponding acyclic program path (uncompressed Whole Program Path), also in megabytes. Acyclic program paths are 4.5–13.7 times larger than bit traces. From this ratio, and given that each intraprocedural acyclic path takes

---

[5] http://sequitur.info/
[6] http://www.cs.dartmouth.edu/~doug/source.html
[7] gcc.gnu.org

95

| Program | APP best | | | APP $l = g = 7$ | WPP |
	l	g	Compr.	Compr.	Compr.
gzip	8	6	5.7	5.7	
vpr	7	7	3.8	3.8	3.3
mcf	6	8	5.1	5.0	6.2
crafty	4	10	4.7	4.4	
parser	6	8	4.6	4.6	4.7
vortex	10	4	121.0	107.3	341.8
bzip2	8	6	18.6	18.5	13.6
twolf	0	14	2.5	2.3	2.5
art	12	2	14.8	13.3	45.1
equake	12	2	12.4	11.5	560.9

Table 2: The APP parameters that achieve the best compression rates, the best compression rates, the compression rates with the best overall parameters, and the WPP compression rates

Program	APP $l = g = 7$ (sec)	WPP (sec)	Ratio
gzip	377	more than 2.5 hours	
vpr	228	5246	23.0
mcf	112	798	7.1
crafty	287	more than 2.5 hours	
parser	77.8	728	9.3
vortex	110	1029	9.3
bzip2	328	4127	12.5
twolf	144	5033	34.9
art	94.1	370	3.9
equake	60.7	864	14.2

Table 3: Time measurements for APP and WPP, and the WPP/APP time ratio.

two bytes, we can find the average length, in edges, of an intraprocedural acyclic path for that particular run.

Acyclic path traces are significantly larger than bit traces, because the average intraprocedural acyclic path consists of eight [3] or fewer [14] edges. Under bit tracing, eight edges are represented in eight bits; this means that if we represent intraprocedural acyclic path identifiers in more than 8 bits, bit tracing is more economical. Eight bits are enough only for 255 path identifiers, so the trace of any program with more than 255 distinct paths is represented in fewer bits with a bit trace than with a path trace. If we are profiling, paths still have an advantage over edges, since edges require instrumentation at every edge, while the optimized path instrumentation can be far leaner.

7.2 Compression

The degree of compression achieved by APP depends primarily on the values of g and l, the number of global and local context bits respectively. We run APP for every pair l, g such that both l and g are even and $l + g \leq 14$, as well as as $l = g = 7$. Table 2 shows the parameters that obtain the best compression and the resulting compression rate. We found that the combination $l = g = 7$ obtained the best compression overall; we report the results in the same table. We also show the result of applying Sequitur to the acyclic program paths of the same execution. In two cases, Sequitur did not finish in the 2.5 hour limit we set.

The best compression for each program is obtained for some combination of l and g where $l + g = 14$, the maximum sum value we experimented with. This suggests that, at least up to 14, more bits of history increase the effectiveness of APP. The compression obtained with $l = g = 7$ is close to the best compression obtained with any l and g where $l + g \leq 14$. This suggests that an even split between the bits dedicated to local and global history is close to the optimal choice, and such a choice can be safely made without significant loss in compression rate.

Overall, APP matches Sequitur's compression. The exceptions are *art*, *vortex*, and *equake*. Two of these programs (*art* and *equake*) are floating point benchmarks, and consist of a few nested loops of straight-line code resulting in a small number of intraprocedural acyclic paths. This is exactly the structure that WPP exploits, and it is not surprising that it outperforms APP. The same holds for the third program, *vortex*, although the loops are implicit. In contrast, APP

shines when many complex paths occur, for example in the compression programs.

7.3 Speed

Table 3 shows the time taken by APP and WPP to compress traces. The measurement for APP was online; that is, APP was built into the subject program. The time for WPP is offline; we stored the acyclic paths, and run Sequitur on them. We subtracted the time it takes to read the acyclic paths from the reported time for Sequitur.

We found that the l and g parameters have little impact on the speed of APP because the program does not need to do any additional work for larger l and g: the core loop is a simple lookup in the three-dimensional table. We report time measurements for $l = g = 7$.

APP is at least three times as fast as WPP, and on average about ten times faster. APP does very little work to update its internal data structures, while Sequitur must manipulate linked lists and a hashtable on every operation.

During decompression, Sequitur has only to unfold the grammar; APP must maintain the model and follow the program's control flow. As a result, while Sequitur decompression speed is much higher than its compression speed (for example, it takes about 100 seconds to decompress the vortex trace), APP's speed is approximately equal to its compression speed. However, we need to note here that the information obtained by the APP decompression is not directly comparable with the WPP decompression. In particular, to retrieve the actual execution trace from an acyclic path, we need to map the path identifiers back to sequences of edges.

7.4 Intraprocedural Arithmetic Program Paths

One of the advantages of APP is that we can instrument and compress individual conditionals. In that case, however, the context of the probability distribution cannot depend on conditionals that the encoder or decoder does not see. This means that to encode individual conditionals, our model must have $g = 0$. However, we can encode blocks of conditionals, for example whole procedures, if we ensure that all the bits of global context come from available conditionals, i.e. conditionals in the same function. We can therefore save and restore the global context upon entry to a function and then utilize only this intraprocedural "global" context. We can do this easily by storing the context in a procedure-local variable. Figure 5 shows how this affects the
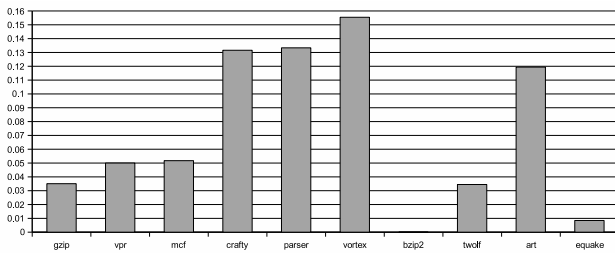
Figure 5: The loss in compression when using intra-procedural history only.

compression rate. The compression loss is under 15%, except for vortex, which has a huge compression rate; and even in this case the loss is under 20%. Static analysis [11] has shown that interprocedural contexts can be helpful in determining the outcome of branches, and therefore we expected some compression loss; the contribution here is quantifying it and using compression as a dynamic way measure interprocedural dependencies.

8. CONCLUSION AND FUTURE WORK

We presented Arithmetic Program Paths, a flexible and memory efficient way to compress program path traces by exploiting the trace structure with the aid of branch-prediction based technique and arithmetic coding. In general, arithmetic program paths compress as well as state-of-the-art systems, but within a smaller memory footprint, and an order of magnitude faster.

We plan to extend this work along two major axes. The first axis is timestamping. Since program traces are long, it can be useful to have random access along the time dimension [33]. This can be achieved by periodically storing the state of the APP as a timestamp. The state of the APP consists of the subject program's stack, the current conditional, the state of the arithmetic coder (the codeword interval) and the model. All but the model are of moderate size. There are several ways to store the model effectively or avoiding storing it altogether: discovering the model offline, by running the subject program on representative test cases; re-initializing or simplifying the model before exporting the timestamp; or storing the model incrementally. All of these potential solutions will incur some penalty on the compression rate or speed; we would like to investigate this experimentally.

Second, we plan to exploit the compression rate to understand program behavior. For example, for some programs, limiting the global context to only intraprocedural information has a big effect on APP. Some of the interprocedural effect can be discovered statically [11]. Contrasting the static with the dynamic results promises insight as to the dynamic nature of interprocedural effects.

9. REFERENCES

[1] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1990.

[2] Glenn Ammons, Rastislav Bodik, and James Larus. Mining specifications. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002.

[3] Thomas Ball and James R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 46–57, 1996.

[4] M. Burtscher. Vpc3: A fast and effective trace-compression algorithm. In *Joint International Conference on Measurement and Modeling of Computer Systems*, pages 167–176, June 2004.

[5] M. Burtscher and M. Jeeradit. Compressing extended program traces using value predictors. *International Conference on Parallel Architectures and Compilation Techniques*, pages 159–169, 2003.

[6] John G. Cleary and Ian H. Witten. A comparison of enumerative and adaptive codes. *IEEE Transactions on Information Theory*, 30(2):306–315, March 1984.

[7] Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: Less is more. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[8] Rajiv Gupta, Mary Lou Soffa, and John Howard. Hybrid slicing: Integrating dynamic information with static analysis. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, October 1995.

[9] Pascal Van Hentenryck. Incremental Constraint Satisfaction in Logic Programming. In *Seventh International Conference on Logic Programming*, Jerusalem, Israel, June 1990.

[10] D.A. Huffman. A method for the construction of minimum redundancy codes. In *Proceedings of IRE*, volume 40, pages 1098 – 1101, 1952.

[11] Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa. Interprocedural conditional branch elimination. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1997.

[12] G. G. Langdon. Introduction to arithmetic coding. *IBM Journal of Research and Development*, 1984.

[13] J. Larus. Abstract execution: a technique for efficiently tracing programs. *Software — Practice& Experience*, 20, 1990.

[14] James R. Larus. Whole program paths. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 259–269, 1999.

[15] Zhijian Lu, John Lach, Mircea R. Stan, and Kevin Skadron. Alloyed branch history: Combining global and local branch history for robust performance. *International Journal of Parallel Programming*, 31(2):137–177, April 2003.

[16] Scott McFarling. Combining branch predictors. Technical Note TN-36, Digital Equipment Corporation Western Research Laboratory (WRL), June 1993.

[17] David Melski and Thomas Reps. Interprocedural path profiling. Technical Report CS-TR-1998-1382, University of Wisconsin, Madison, September 1998.

[18] A. Moffat, R. Neal, and I.H. Witten. Arithmetic coding revisited. In *IEEE Data Compression Conference*, pages 202–211, 1995.

[19] Ravi Nair. Branch behavior on the IBM RS/6000. Technical Report 17859, IBM Thomas J. Watson Research Center, 1992.

[20] George C. Necula, Scott McPeak, S.P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *International Conference on Compiler Construction*, April 2002.

[21] Craig G. Nevill-Manning and Ian H. Witten. Linear-time, incremental hierarchy inference for compression. In *DCC: Data Compression Conference.* IEEE Computer Society TCC, 1997.

[22] R. Pasco. *Source Coding Algorithms for fast data compression.* PhD thesis, Dept. of EE, Stanford University, Stanford, CA, 1976.

[23] Andrew Pleszkun. Techniques for compressing program address traces. *XXVII Annual IEEE/ACM Symposium on Microarchitecture*, pages 32–40, 1994.

[24] Steven P. Reiss. Trace-based debugging. In Peter Fritzon, editor, *Automated and Algorithmic Debugging, First International Workshop, AADEBUG'93*, volume 749 of *Lecture Notes in Computer Science*, pages 305–314. Springer, 3–5 May 1993.

[25] Steven P. Reiss and Manos Renieris. Encoding program executions. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 221–230, 2003.

[26] J.J. Risannen. Arithmetic codings as number representations. In *Acta Polytech. Scand. Math.*, 1979.

[27] Claude E. Shannon. A mathematical theory of communication. Technical report, Bell Systems Technical Journal, 1948.

[28] John Paul Shen and Mikko H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors.* McGraw-Hill, 2005.

[29] James E. Smith. A study of branch prediction strategies. In *ISCA '81: Proceedings of the 8th annual symposium on Computer Architecture*, pages 135–148. IEEE Computer Society Press, 1981.

[30] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[31] I.H. Witten, R. Neal, and J. Cleary. Arithmetic coding for data compression. *Comm. of the ACM*, pages 520–540, 1987.

[32] Tse-Yu Yeh and Yale N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *ISCA*, pages 257–266, 1993.

[33] Y. Zhang and R. Gupta. Timestamped whole program paths. *Conference on Programming Language Design and Implementation*, 2001.