Branch Trace Compression for Snapshot-Based Simulation

Kenneth C. Barr and Krste Asanović
MIT Computer Science and Artificial Intelligence Laboratory
The Stata Center, 32 Vassar Street, Cambridge, MA 02139
{kbarr, krste}@csail.mit.edu

Abstract

We present a scheme to compress branch trace information for use in snapshot-based microarchitecture simulation. The compressed trace can be used to warm any arbitrary branch predictor's state before detailed microarchitecture simulation of the snapshot. We show that compressed branch traces require less space than snapshots of concrete predictor state. Our branch-predictor based compression (BPC) technique uses a software branch predictor to provide an accurate model of the input branch trace, requiring only mispredictions to be stored in the compressed trace file. The decompressor constructs a matching software branch predictor to help reconstruct the original branch trace from the record of mispredictions. Evaluations using traces from the Journal of ILP branch predictor competition show we achieve compression rates of 0.013–0.72 bits/branch (depending on workload), which is up to 210× better than gzip; up to 52× better than the best general-purpose compression techniques; and up to 4.4× better than recently published, more general trace compression techniques. Moreover, BPC-compressed traces can be decompressed in less time than corresponding traces compressed with other methods.

1. Introduction

As full-system simulation becomes more popular and workloads become longer, methodologies such as statistical simulation and phase detection have been proposed to produce reliable performance analysis in a small amount of time [12, 21, 22, 30]. When such techniques are adopted, the bulk of simulation time is spent fast-forwarding the simulator to relevant points in a program rather than performing detailed cycle-accurate simulation. To amortize the cost of lengthy fast-forwarding, snapshots containing architectural state can be captured at each sample point [2, 29]. The snapshots can then be used to initialize different machine configurations without repeating the fast-forwarding. Microarchitectural state can be reconstructed using a "detailed warming" phase before results are gathered at each sample point, but if the microarchitectural state is large, such as for caches and branch predictors, the time required for detailed warming can be prohibitive.

Microarchitectural state can also be captured in the snapshot, but this will then require regeneration of the snapshot every time a microarchitectural feature is modified. Ideally, the snapshots should be microarchitecture independent, to support microarchitectural exploration with a standard set of stored snapshots. For caches, various microarchitecture-independent snapshot schemes have been proposed, which take advantage of the simple mapping of memory addresses to cache sets [1, 13, 27, 28]. Branch predictors, however, are much more difficult to handle in the same way, as the common use of global branch histories smears the effect of a single branch address across many locations in a branch predictor. One possibility is to store microarchitectural state snapshots for a set of of potential branch predictors, but this limits flexibility and increases snapshot size, particularly when many samples are taken of a long-running multiprocessor application. We explore an alternative approach in this paper, which is to store a compressed version of the complete branch trace in the snapshot. This approach is microarchitecture-independent because any branch predictor can be initialized before detailed simulation begins by uncompressing and replaying the branch trace.

The main contribution of our paper is a branch predictor-based compression (BPC) scheme, which exploits a software branch predictor in the compressor and decompressor to reduce the size of the compressed branch trace snapshot. We show that when BPC is used, the snapshot library can require less space than one which stores just a single concrete predictor configuration, *and* it allows us to simulate *any* sort of branch predictor.

We describe the structure of our BPC compressor in Section 2. In Section 3, we examine the improvement of our technique versus general-purpose compressors and snapshots, and we show how it scales with program size in terms of storage and performance. Section 4 notes related work in compression and simulation acceleration, and Section 5 concludes.

2. Design of a branch predictor-based trace compressor

In general, lossless data compression can be partitioned into two phases: modeling and coding. The modeling phase attempts to predict the data symbols. For each symbol in the input text, the compressor expresses any differences from the model. The coding phase creates concise codewords to represent these differences in as few bits as possible. BPC uses a

collection of *internal predictors* to create an accurate, adaptive model of branch behavior. We delegate the coding step to a general-purpose compressor.

To model the direction and targets of branches in a trace, we can draw on years of research in high accuracy hardware branch predictors. When using branch predictors as models in software, we have two obvious advantages over hardware predictors. First, the severe constraints that usually apply to branch prediction table sizes disappear; second, a fast functional simulator (which completes the execution of an entire instruction before proceeding) can provide oracle information to the predictor such as computed branch targets and directions. We use the accurate predictions from software models to reduce the amount of information passed to the coder. When the model can predict many branches in a row, we do not have to include these branches in the compressor output; we only have to express the fact that the information is contained in the model.

2.1. Structure

Figure 1 shows the different components in our system. A benchmark is simulated with a fast functional simulator, and information about branches is passed to the BPC Compressor. The BPC Compressor processes the branch stream, filters it through a general-purpose compressor, and creates a compressed trace on disk. We will show momentarily how the BPC Compressor can improve its compression ratios by using its own output as input to compress the next branch.

We define a *concrete branch predictor* to be a predictor with certain fixed parameters. These parameters may include size of global history, number of branch target buffer (BTB) entries, indexing functions, et cetera. To evaluate the performance of various concrete branch predictors, we retrieve the compressed trace from disk, remove the general-purpose compression layer, and process it with the BPC Decompressor. The structure of the decompressor is identical to that of the compressor. The output of the BPC Decompressor is used to update state in each concrete predictor used in the experiment. Branches later in the trace will overwrite entries in the concrete predictor according to its policies.

The particular collection of internal predictors has nothing to do with the concrete branch predictors that BPC will warm. The implementation of BPC merely uses predictors to aid compression of the complete branch trace which, by its nature, can be used to fill *any* branch predictor with state based on information in the trace. Furthermore, the precise construction of a BPC scheme is up to the implementor who may choose to sacrifice compression for speed and simplicity or vice versa. We merely describe what appears to be a happy medium.

Note this implementation differs from other proposed value predictor-based compression (VPC) techniques, which feed several predictors in parallel and emit a code indicating which predictor is correct [4, 26]. We have found that, for our datasets of branch traces, indicating in the output stream which one of a set of predictors has succeeded results in a stream of indicator codes which is itself difficult to compress.

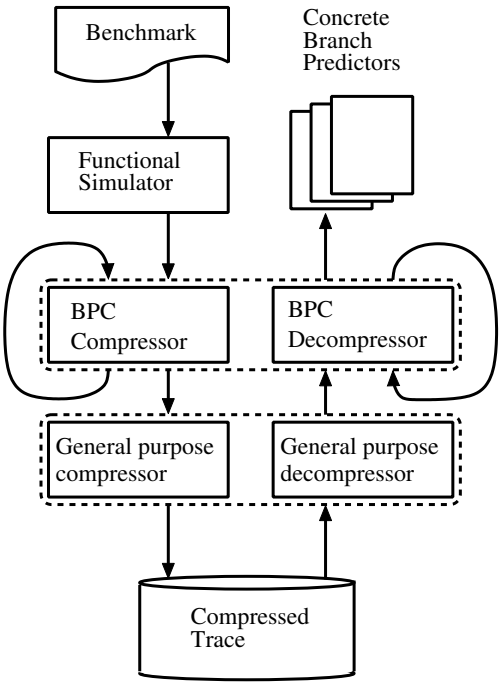

Figure 1. System diagram. The compressed trace is stored on disk until it is needed to reconstruct concrete branch predictors.

2.2. Branch notation and representation

Before getting into the details of the compressor, we describe the information stored in the compressed trace and introduce some notation used in this paper.

Using the Championship Branch Prediction (CBP) framework [26], the uncompressed branch traces in our study consist of fixed-length *branch records* as shown in Table 1. Each branch record contains the current instruction address, the fall-through instruction address (not obvious in CISC ISAs), the branch target, and type (not mutually exclusive) which indicates whether the branch is a call, return, indirect, or conditional. The branch records are generated by a functional simulator which can resolve the branch target and provide a taken/not-taken bit. The taken bit is stored in a one-byte field to facilitate compression via popular bytewise algorithms such as gzip [11], bzip2 [23], or PPMd [24].

Table 1. Format of branch records.

field				size (Bytes)
instruction address				4
fall-through instruction address				4
branch target address				4
taken				1
is_indirect	is_conditional	is_call	is_return	1

Rather than predicting the direction and target of the current branch, B_n, as in a hardware direction predictor and branch target buffer (BTB), we predict information about the *next* branch, B_{n+1}. We denote the actual branch stream as $B_{0..k}$ and predicted branches as $\beta_{0..k}$. If the predictor proves correct ($\beta_{n+1} = B_{n+1}$) we concisely note that fact and need not provide the entire B_{n+1} branch record. Furthermore, we use β_{n+1} to produce $\beta_{n+2..n+i}$ for as large an i as possible. This allows us to use a single symbol to inform the decompressor that i chained predictions will be correct.

Figure 2 depicts an example using this notation. Given B_n we must provide information about β_{n+1}. In this case, we have predicted β_{n+1} to be not-taken with a specific fall-through address. If these are correct predictions, BPC can continue by chaining: using β_{n+1} as input to request predictions about β_{n+2}. The longer the chain of correct predictions, the less information has to be written by the compressor.

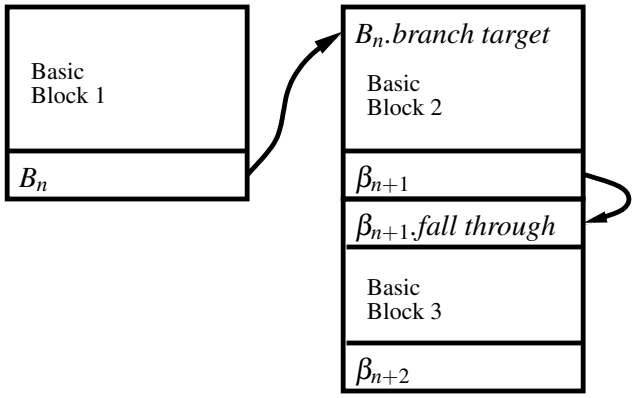

Figure 2. Graphic depiction of our notation.

The output of the compressor is a list of pairs. The first element indicates the number of correct predictions that can be made beginning with a given branch; the second element contains the data for the branch that *cannot* be predicted. An example output is shown in Table 2. As in most branch traces, the example shows that the first few branches cannot be predicted and must be transmitted in their entirety. Eventually the compressor outputs B_{10} and uses B_{10} as an input to its internal predictors, coming up with a prediction, β_{11}. Comparing β_{11} to the next actual branch, B_{11}, a match is detected. This process continues until the internal predictors fail to guess the incoming branch (at B_{24}). Thus, we output "13" to indicate that, by chaining predictor output to the next prediction's input, 13 branches in a row will be predicted correctly, but $\beta_{24} \neq B_{24}$. We emit B_{24} and repeat the process.

We store the output in two separate files and use a general-purpose compressor (such as gzip or bzip2) to catch patterns that we have missed and to encode the reduced set of symbols.

Table 2. Example compressor output.

skip amount	branch record
0	B_0
0	B_1
0	B_2
...	...
0	B_{10}
13	B_{24}

2.3. Algorithm and implementation details

The internal structure of a BPC Compressor is shown in Figure 3. Each box corresponds to one predictor. When multiple predictors are present at a stage, only *one* is consulted. In BPC, the criteria for choosing a predictor stems from branch *type* which expresses characteristics of the branch such as whether it is a return instruction or if it is conditional. The details of how type determines predictor selection are explained in Sections 2.3.3 and 2.3.4.

The description below refers to the steady-state operation. We do not describe handling of the first and last branch, nor do we detail the resetting of the *skip* counter. These corner cases are addressed in our code. We use `diff` to ensure a compressed trace can be uncompressed correctly.

Initially, a known address, target, and taken bit of B_n are received from the functional simulator and used to predict the address of β_{n+1}. This address is used to look up static information about β_{n+1} including its fall-through address, type, and target. In the absence of context switches or self-modifying code, the branch address corresponds directly with a type and fall-through address. If the branch target is not computed, a given branch address always has the same branch target. The type prediction helps the direction predictor decide whether β_{n+1} is a taken branch. The type also helps the target predictor make accurate predictions. Once components of β_{n+1} have been predicted, it can be used to generate a prediction for β_{n+2} and so on. Before continuing, the predictors are updated with correct data for B_{n+1} provided by the simulator.

2.3.1. Predicting the next branch address. Normally, branch targets must be predicted, but in the case of BPC, they are known by the functional simulator. Instead, we must predict the address of the next *branch*. Since the next branch seen by a program is the first branch to appear after reaching a target, knowing the target allows us to know the next branch. If the branch is not taken, the next branch should appear shortly a few instructions later. This prediction will be 100% correct unless we are faced with self-modifying code or process swaps.

If B_n is a taken branch, we use a simple mapping of current branch target to next branch address. This map can be considered a Last-1-Value predictor or a cache. Our map is implemented with a 256K-entry hash table. The hash tables and fixed sized predictors of BPC provide $O(1)$ read and write time with

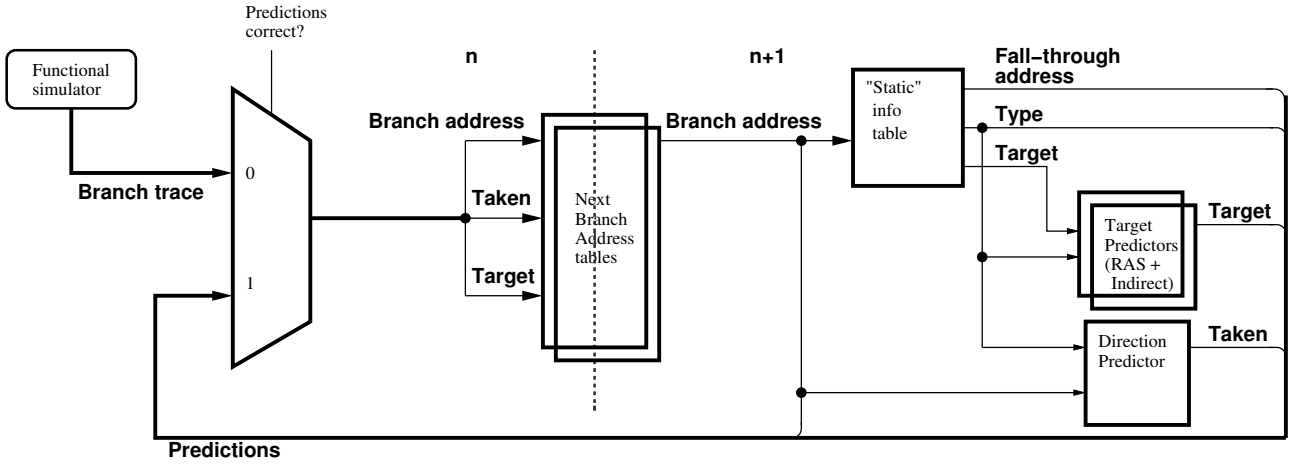

Figure 3. Prediction flow used during branch trace compression. Input left of the dashed line is from the current branch, B_n. To the right of the dashed line are predictions for the next branch, β_{n+1}.

respect to trace length. Since the hash table merely provides predictions, it need not detect or avoid collisions. This property permits a fast hash function and immediate insertion/retrieval, but we choose to use linear probing to avoid collisions and achieve higher prediction accuracy.

If B_n is not taken, we use a separate table indexed by current branch address to reduce aliasing. By using two tables we insure that taken and not-taken instances of the same branch do not stomp on each other's next address mapping.

Recall that BPC is benefiting from the oracle information provided by the functional simulator. Hardware target predictors are accessed early in the cycle before the instruction has been decoded and resolved. Here, the simulator has produced B_n.*taken* and B_n.*target* which it uses to select and index the maps.

2.3.2. Predicting the next branch's static information. The compressor looks up β_{n+1}.*branch address* in a hash table to find β_{n+1}.*type*, β_{n+1}.*fall-through address*, and a potential β_{n+1}.*target*. Note that this target may be overridden if the branch type indicates an indirect branch or return instruction. The lookup table is implemented as in Section 2.2.1.

2.3.3. Predicting the next branch's direction. If the (predicted) type indicates the next branch is conditional, we look up its direction in a direction predictor. Recall that our software predictors are not constrained by area or cycle time as in a hardware predictor. Thus, we chose a large variation of the 21264-style tournament predictor [15]. We XOR the program counter with a 16 bit global branch history to access a global set of two-bit counters, and use 2^{16} local histories (each 16 bits long) to access one of 2^{16} three-bit counters. A chooser with 2^{16} two-bit counters learns the best-performing predictor for each PC/History combination. This represents 1.44 Mbits of state, much more than one would expect in a real machine. Branches predicted to be non-conditional are marked Taken.

2.3.4. Predicting the next branch's target. If the next branch is a return, we use a 512-deep return address stack to set its target. This extreme depth was necessary to prevent stack overflows in some of our traces that would have hidden easily-predicted return addresses.

If the next branch is a non-return indirect branch, we use a large filtered predictor to guess the target [9]. We introduce a 32 K-entry BTB leaky filter in front of a path-based indirect predictor. The path-based predictor is a 2^{20} entry version of the predictor provided by the Championship Branch Prediction contest [26]. It has a PAg structure and uses the last four targets as part of the index function. The filter quickly learns targets of monomorphic branches, reducing the cold-start effect and leaving more room in the second-stage, path-based predictor.

If the next branch is neither a return nor an ordinary indirect branch, we set the target equal to the last target found in Section 2.3.2.

2.3.5. Emitting the output stream and continuing. The β_{n+1} structure created thus far is compared with the actual next branch, B_{n+1}. If they match, we increment a *skip* counter; if not, we emit $< skip, B_{n+1} >$. To keep a fixed-length output record, we do not allow *skip* to grow past a threshold (e.g., a limit of 2^{16} allows the skip value to fit in two bytes).

Before repeating, all predictors are updated with the correct information: for each B_n, the instruction address tables are indexed by B_n's address or target address and updated with B_{n+1}.*instruction address*, while the remaining predictors are indexed by B_{n+1}.*instruction address* and updated with B_{n+1}'s resolved target, taken bit, fall-through address, and type. Finally, we increment n and repeat the above steps.

2.3.6. Entropy Coding. The output of the BPC compressor is significantly smaller than the original branch trace, but better results are possible by employing a general-purpose compressor such as gzip, bzip2, or PPMd. These highly tuned com-

pressors are sometimes able to capture patterns missed by BPC and use Huffman or arithmetic coding to compress the output further.

2.3.7. Decompression Algorithm. The decompressor must read from these files and output the original branch trace one branch at a time and in the correct order. The BPC decompression process uses the same structures described in Sections 2.3.1–2.3.5 so it can be described quickly. As above, we assume a steady state where we have already read B_n.

After reversing the general-purpose compression, the decompressor first reads from the *skip amount* file. If the skip amount is zero, it emits B_{n+1} as found in the *branch record* file, and updates its internal predictors using B_n and B_{n+1}.

If it encounters a non-zero skip amount, it uses previous branch information to produce the next branch. In other words, to emit B_{n+1} it queries its internal predictors with B_n and outputs the address, fall-through, target, type, and taken information contained in the internal predictors. Next, the skip amount is decremented, B_{n+1} becomes the current branch (B_n), and the process repeats. Eventually, the skip amount reaches 0, and the next branch must be fetched from the input file rather than emitted by the predictors.

As the decompressor updates its internal predictors using the same rules as the compressor, the state matches at every branch, and the decompressor is guaranteed to produce correct predictions during the indicated skip intervals. The structure of the decompressor is identical to that of the compressor, so decompression proceeds in roughly the same time as compression.

Recall that the motivation for compressing a branch trace is to replace concrete branch predictor snapshots for sampling-based simulation. By piping the output of our decompressor into a concrete branch predictor model, the model becomes warmed-up with exactly the same state it would have contained had it been in use all along. Furthermore, the decompressed branch stream can be directed into *multiple* concrete branch predictors so that each may be evaluated during detailed simulation. After each branch is examined it may be discarded to minimize disk usage. Since the trace is generated with a fast non-speculative model, the predictors do not capture wrong-path effects, but the resulting bias has been shown to be acceptable [30].

3. Evaluation

Our simulation framework is based on the CBP competition trace reader which provides static and dynamic information about each branch in its trace suite. The trace suite consists of 20 traces from four categories: integer, floating point, server, and multimedia. Each trace contains approximately 30 million instructions comprising both user and system activity [26] and exhibiting a wide range of characteristics in terms of branch frequency and predictability as shown in Table 3. The traces are used to drive predictors from the competition as well as custom models. Columns labeled CBP show the direction accuracy and indirect target accuracy of the predictors used in the Championship Branch Prediction trace reader: a gshare predictor with a 14-bit global history register, and an indirect target predictor in a PAg configuration with 2^{10} entries and a path-length of 4 (bits from the past four targets are hashed with the program counter to index into a target table). The BPC column shows the decreased misprediction rate available to BPC with the configuration described in Section 2.3.3 and Section 2.3.4.

Using this framework we will show that BPC provides an excellent level of compression. Not only does a compressed trace require less space than compressed snapshots, but a BPC-compressed trace is smaller and faster to decompress than other compression techniques.

3.1. Compression ratio

Figure 4 shows the compression ratio resulting from various methods of branch trace compression for each trace. Traces were run to completion with snapshots taken every 1M instructions. This sampling interval was found to produce good results on Spec benchmarks [30]. Each trace provides enough branches for 29 snapshots. We report bits-per-branch (rather than absolute file size or ratio of original size to new size) so that our results are independent from the representation of branch records in the input file. From left-to-right we see compression ratios for general-purpose compressors; compressed concrete snapshots; VPC (a similar work which is discussed in Section 3.4); and BPC as described in this paper. We use the suffix *+comp* to denote the general-purpose second-stage compressor.

While slower, bzip2 and PPMd give astonishingly good results on raw trace files composed of fixed-length records. In fact, these general-purpose compressors use algorithms that have a more predictive nature than the dictionary-based gzip.

The three bars labeled "concrete" show the size of a snapshot containing a single branch predictor roughly approximating that of the Pentium 4: a 4-way, 4096 entry BTB to predict targets and a 16-bit gshare predictor to predict directions [20]. Together the uncompressed size of the concrete predictor is 43.6 KB, however we use a bytewise representation and store a 97 KB snapshot as it is more amenable to compression than a bitwise representation – up to 20% smaller in some cases. The figure shows the size of bytewise snapshots after compression with gzip, bzip2, and PPMd.

The state of a given branch predictor (a *concrete snapshot* in our terminology) has constant size of q bytes. However, to have m predictors warmed-up at each of n detailed sample points (multiple short samples are desired to capture whole-program behavior), one must store mn q-byte snapshots. Concrete snapshots are hard to compress so p, the size of q after compression, is roughly constant across snapshots. Since a snapshot is needed for every sample period, we consider the cumulative snapshot size: mnp. This cumulative snapshot grows with m and n. In fact, it grows *faster* than a BPC-

Table 3. Characteristics of traces. Note that indirect branches refer to those branches not already classified as Calls or Returns. Unconditional branches are those that remain after classifying indirects, calls, and returns. Columns may not sum to 100% due to rounding. For a description of the BPC and CBP predictors, please see text of Sections 2 and 3 respectively.

Name	Branches	Insts/Branch	Cond-itional	Return	Call	Indirect	Uncond-itional	CBP Direction	BPC Direction	CBP Indirect Target	BPC Indirect Target
	(Millions)		(percent of total)					(Mispred. Rate)		(Mispred. Rate)	
FP-1	2.6	11.3	84.6	5.5	5.5	0.0	4.4	0.051	0.039	0.314	0.288
FP-2	1.8	16.3	99.3	0.0	0.0	0.0	0.6	0.018	0.017	0.317	0.303
FP-3	1.6	18.7	98.3	0.4	0.4	0.0	0.9	0.009	0.008	0.286	0.277
FP-4	0.9	32.0	97.2	0.9	0.9	0.0	1.1	0.010	0.010	0.251	0.241
FP-5	2.7	10.8	89.0	4.6	4.6	0.0	1.8	0.010	0.004	0.598	0.563
INT-1	5.0	5.9	83.9	4.6	4.6	0.0	7.0	0.053	0.049	0.362	0.337
INT-2	3.7	8.0	78.1	6.2	6.2	0.8	8.7	0.078	0.074	0.597	0.526
INT-3	4.1	7.1	91.2	0.7	0.7	0.0	7.4	0.106	0.094	0.313	0.285
INT-4	2.4	12.1	85.1	5.8	5.8	0.0	3.3	0.036	0.032	0.009	0.008
INT-5	3.8	7.7	98.3	0.5	0.5	0.2	0.6	0.005	0.003	0.285	0.250
MM-1	2.8	10.6	80.1	5.2	5.2	0.0	9.6	0.099	0.108	0.001	0.001
MM-2	4.2	7.0	90.4	2.6	2.6	1.7	2.7	0.079	0.079	0.015	0.011
MM-3	5.0	6.0	60.9	16.7	16.7	0.1	5.7	0.030	0.014	0.114	0.101
MM-4	5.1	5.8	95.9	1.5	1.5	0.2	0.9	0.011	0.011	0.053	0.046
MM-5	3.4	8.7	75.3	8.9	8.9	2.6	4.3	0.067	0.055	0.172	0.062
SERV-1	5.6	5.3	65.3	12.3	12.4	0.4	9.6	0.040	0.021	0.357	0.024
SERV-2	5.4	5.4	65.0	12.3	12.3	0.4	10.0	0.043	0.023	0.377	0.026
SERV-3	5.4	5.5	71.1	8.3	8.3	0.2	12.0	0.045	0.037	0.113	0.057
SERV-4	6.3	4.7	67.7	10.3	10.3	0.3	11.3	0.040	0.026	0.242	0.023
SERV-5	6.4	4.6	66.9	10.4	10.4	0.3	12.0	0.040	0.025	0.258	0.019

compressed branch trace even for reasonable p and $m = 1$. Note that combining the concrete snapshots before compression provides context which is helpful for compression, but neither reflects typical snapshot usage models nor approaches the trace compression ratios.

On average, BPC+PPMd provides a 3.4×, 2.9×, and 2.7× savings over stored predictors compressed with gzip, bzip2, and PPMd respectively. When broken down by workload, the savings of BPC+PPMd over concrete+PPMd ranges from 2.0× (integer) to 5.6× (floating point). Using BPC+PPMd rather than concrete predictors compressed with gzip, bzip2, and PPMd, translates to an absolute savings (assuming 20 traces, 1 billion instructions per trace, and an average of 7.5 instructions per branch) of 257MB, 207MB, and 182MB respectively. Note that this represents the *lower bound* of savings with BPC: if one wishes to study m branch predictors of size $P = \sum_{i=1}^{m} p_i$, the size of the concrete snapshot will grow with mnP, while the BPC trace supports any set of predictors at its current size.

From these results, we note that predictive compressors (bzip2, PPMd, and BPC) outperform dictionary-based compressors in all cases, often drastically. Furthermore, BPC+bzip2 outperforms pure bzip2 in all cases. To be fair, we considered PPMd, a fast implementation of the PPM algorithm which feeds to an arithmetic encoder and tends to produce better compression ratios than bzip2 in roughly equal time. While PPM-based methods are often discounted due to their slow speed, we found PPMd to perform faster than bzip2 for our source data. Of course, a mild speed penalty during the compression phases could be accepted as snapshot generation occurs just once. We capped PPMd's memory to 32 MB and used an order-14 model corresponding to the number of bytes-per-record in the raw trace file; this corresponds to a 1st-order model at the branch record granularity.

We exceed stand-alone PPMd compression in 15/20 cases. In a sense, BPC is similar to the Markov modeling used by PPM. However, the additional context (e.g., long global histories and deep return address stack) usually allows us to predict better than the simpler model constructed by PPMd. In the cases where PPMd does better, we may be able to tease out additional improvement through the use of stride predictors or improved direction and indirect branch predictors.

Figure 5 shows the length of correct prediction chains and helps explain the success of BPC. Recall that long chains are represented by a single number in the *skip amount* output file and a single branch in the *branch record* file. These histograms show *sums* rather than average skip counts for each application domain, and we normalize to total number of branches to allow cross-domain comparison. In terms of total branches, we remove over 90% of branches in all cases and we remove over 95% in all but four cases: integer and multimedia are the most troublesome due to lower accuracy in the direction predictor.

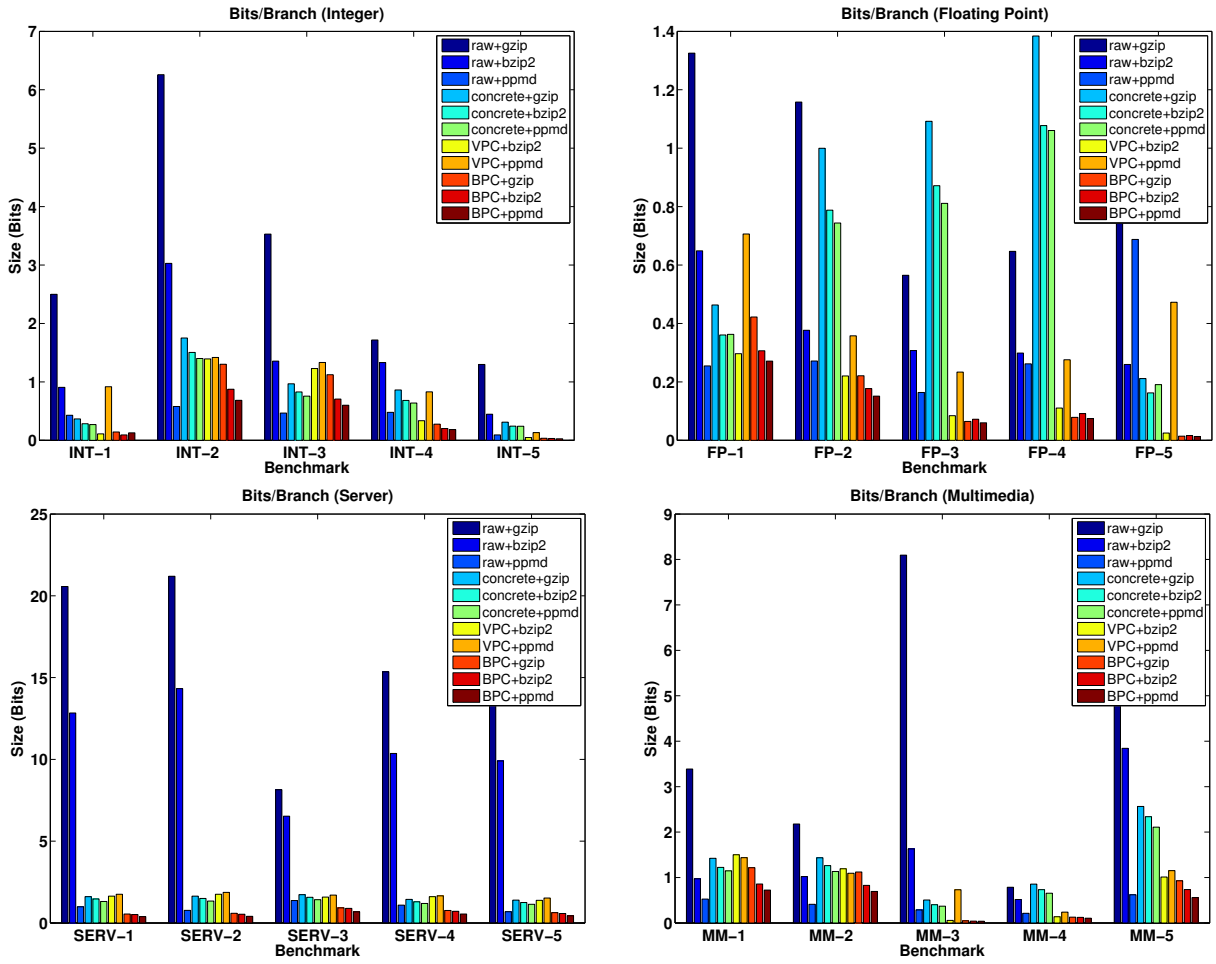

Figure 4. Compressed size (bits/branch). Different y-axis scales used for visibility.

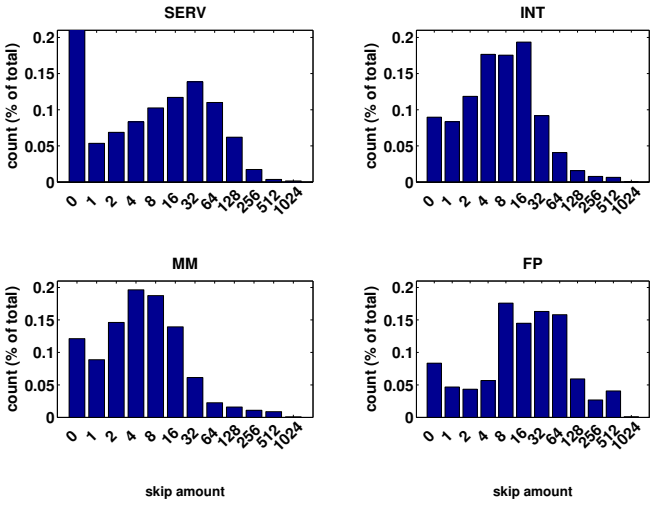

Figure 5. Skip amount frequency. Large skip amounts indicate chains of correct predictions. Bucket *i*, labeled *b[i]*, tallies skip amounts greater than or equal to *b[i]*, but less than *b[i+1]*.

3.2. Scaling

How can we be certain that the compression ratios observed on this short trace will carry through an entire program execution? We extrapolate from the data shown in Figure 6 which shows how storage requirements scale over time. Since a single compressed trace suffices to represent all branches in the program, we report the current size of the compressed trace at every 1M instructions. For the concrete snapshots, we report the cumulative snapshot size at the given instant. Note that Figure 4 is merely the final data point of Figure 6 divided by the number of branches observed and multiplied by 8 bits.

As the program begins, the concrete predictors are largely unused and the easily compressed. Thus, their total size is less than a compressed trace. As the program progresses, the concrete predictors are harder to compress and make a dramatic contribution to the overall snapshot size. For all workloads, trace compression scales better than storing concrete predictors. In 15/20 cases, BPC compression fares better than PPMd compression of a raw branch trace; in two other cases (INT2 and FP1) it is competitive or nearly equal, leaving three cases (MM1, MM2, and INT3) in which PPMd outperforms

BPC+PPMd. The server benchmarks present an interesting challenge to the compression techniques. In general, these workloads contain branches which are harder to predict and phase changes which are more pronounced. BPC, with its hardware-style internal branch predictors, is more suited to quick adaptation than PPMd which uses more generic prediction. When returning to a phase, BPC's large tables and long history allow better prediction than PPMd, which must adjust its probability models as new inputs are seen; when old inputs return, the model's representation of old data is less optimal.

The figure shows trends developing in the early stages of execution that should continue throughout the program. A trace compressed with BPC will grow slowly as new static branches appear, but reoccurrence of old branches will be easily predicted and concisely expressed (unless purged from the model). Storage of concrete snapshots grows with *mnP* as discussed in Section 3.1.

3.3. Timing

We have shown the storage advantages of trace-based reconstruction versus snapshot-based reconstruction, but we must show that the time required to compress and decompress the data does not outweigh the space savings. In the case of BPC or BPC+general purpose *compression*, the cost is negligible. BPC requires simple predictors and tables which adds little time to functional simulation. The second-stage, general-purpose compressors (gzip, bzip2, and PPMd) are highly optimized and use fixed-size tables, so they remain fast throughout the compression process. Furthermore, compression is performed once, so the creation time of a microarchitectural snapshot can be amortized over many detailed simulations. We no longer have to guess likely configurations, fix a maximum size, or regenerate a snapshot to reflect a microarchitectural change.

Decompression speed is more important. We presume a parallel methodology in which independent snapshots are produced and used to warm up state for detailed samples on multiple machines. In such a situation, runtime is limited by the time to warm the final sample in program order. When working with a non-random-access compressed trace such as BPC, the warming for the final sample in the program requires examining every branch in the trace. While this is much slower than directly loading a snapshot of microarchitectural state, it is much faster than functional simulation. Intuitively, warming via branch trace decompression can be faster than functional simulation: not only are there many fewer branches than total instructions, but for each branch, only a few table updates are required rather than an entire decode and execute phase. We have traded some speed for lots of flexibility while remaining several times faster than traditional functional branch predictor warming. On average, BPC adds 48 seconds for every billion instructions in the program on our test platform.

Table 4 gives an estimate of the additional time needed to use each compression scheme. The times were collected on a Pentium 4 running at 3 GHz. BPC, VPC and PPMd were compiled with gcc 3.4.3 -O3, and vendor-installed versions of gzip and bzip2 are used. Timing information is the sum of user and system time reported by `/usr/bin/time`. We require each application to write its data to stdout, but redirect this output to `/dev/null`. For VPC, we modify the generated code so that 2nd-stage compression may be performed in a separate step; the sum reported in the table may be slightly slower than had the 2nd-stage compression been performed inline, but it allows us to examine alternative 2nd-stage compressors. sim-bpred is the branch predictor driver distributed with the popular Simplescalar toolset [3]. We run Spec2000 benchmarks using Minnespec datasets [16] with sim-bpred's static not-taken predictor to show how quickly a fast functional simulator can decode and identify branch instructions. Note that the SpecFP average speed is hurt by several benchmarks with a very small percentage of control instructions (e.g., 50 or 120 instructions per branch); while it is a representative average, it is difficult to compare directly with the CBP traces which have closer to ≈ 15 instructions per branch.

Table 4. Performance of BPC and general purpose decompressors. Table shows millions of branches decompressed per second (harmonic means).

	SERV	INT	MM	FP	average
gzip	7.27	17.71	15.68	20.23	13.02
bzip2	0.79	0.67	0.71	0.65	0.70
PPMd	0.81	1.12	1.14	1.30	1.06
VPC+bzip2	1.29	1.90	2.03	2.47	1.82
VPC+PPMd	0.95	1.43	1.46	1.68	1.32
BPC+PPMd	2.23	3.18	2.98	4.10	2.98
sim-bpred		1.09		0.34	0.50

Our original BPC implementations used state-of-the-art perceptron predictors and the standard template library (STL) which dominated runtime. The current implementation, which uses a large tournament predictor and no STL, strikes a balance between speed and compression. The table shows that while the impressive compression ratios observed in Figure 4 do not come for free, one can still obtain decompression speeds that surpass an optimized simulator. Not only is BPC faster than a fast RISC functional simulator, we note that the BPC rate will remain constant while functional simulation becomes slower as support for CISC instructions and full system simulation is added.

We see that PPMd performs better than the more common bzip2 when dealing with all categories of branch traces. Combining BPC with PPMd gives us performance up to 3.9× faster than PPMd alone because BPC acts as a filter allowing PPMd to operate on a smaller, simpler trace. The table also shows VPC times for its default 2nd-stage compressor (bzip2) and VPC combined with PPMd. VPC performs best with bzip2, but appears slower than BPC. The speedup is due to a combination of a BPC's simpler hash function; fewer and smaller predictors which may relieve cache pressure; and a more-easily compressed output to the general-purpose compressor.

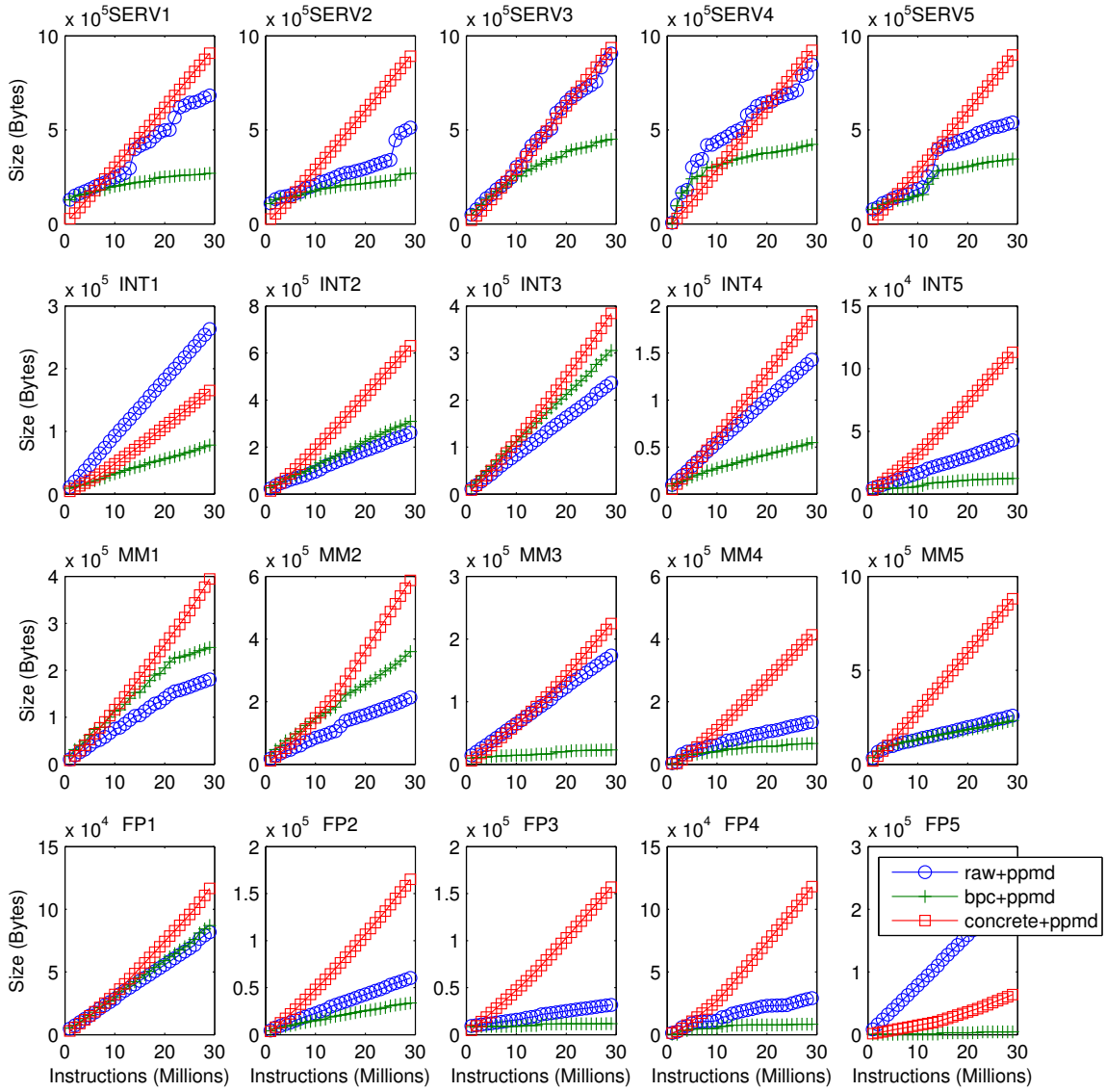

Figure 6. BPC storage requirements grow slower than that of concrete snapshots.

The decompression time is dominated by BPC rather than the general-purpose compression phase, but the fraction varies depending on workload. For example, the small, highly compressed floating point branch traces spend as little as 1.75% of decompression time performing general-purpose decompression, while the server traces require at least 25.0%. Table 5 shows the percentage of time spent performing general-purpose decompression for each class of trace.

3.4. Comparison

Value-predictor based compression (VPC) is a recent advance in trace compression [4]. Its underlying predictors (last-n-value, strided, and (differential) finite context) are more general than the branch direction and target predictors found in BPC. As such, VPC has trouble with branch traces in which branch outcome may only be predicted given a large context. Both predictors must emit incorrectly-predicted branches, but in contrast with BPC, VPC runs several predictors in parallel and emits a code to indicate which predictors should be con-

Table 5. Decompression time is shared between general-purpose decompressor and BPC. Table shows statistics for percent of time spent performing general purpose decompression.

	min	max	mean	st. dev
SERV	25.1%	42.0%	30.8%	7.2%
INT	2.6%	43.0%	23.4%	18.5%
MM	4.0%	43.9%	27.3%	18.2%
FP	1.8%	20.6%	9.1%	7.7%

sulted for each record. When an internal predictor cannot be used, the unpredictable portion of the record is output. Separate output streams are used corresponding to each internal predictor.

To see the improvement possible with specialized predictors, we used TCgen, an automated code generator, to generate a VPC compressor/decompressor pair [6]. We begin as suggested by the developers, by generating code with many predictors (we used 44 of different classes and context lengths); running it on our traces; and refining to include only those predictors that perform best. Paring down the predictors eliminates additional output streams and reduces variability in the correct-predictor index that can negatively effect compression. Eventually we settled on the TCgen specification in Figure 7 which uses 18 predictors, eliminates the simpler last-value predictor, and uses finite-context predictors only where most useful. Figure 4 and Table 4 include data for VPC. We see that BPC compresses branch trace data better than VPC in 19/20 cases (all 20 if we always choose the best 2nd-stage compressor for each) and is between 1.1 and 2.2 times faster. We compressed raw VPC streams with both bzip2 and PPMd to show the effect of the second stage compressor. VPC was tuned for integration with bzip2, and this is evident in the results.

```
TCgen Trace Specification;
0-Bit Header;
32-Bit Field 1={L1=1, L2=131072: DFCM3[2], FCM3[2], FCM1[2]};
32-Bit Field 2={L1=65536, L2= 131072: DFCM3[2]};
8-Bit  Field 3={L1=65536, L2= 131072: DFCM3[2]};
32-Bit Field 4={L1=65536, L2= 131072: DFCM3[2], FCM1[2]};
8-Bit  Field 5={L1=65536, L2= 131072: DFCM3[2], FCM1[2]};
PC = Field 1;
```

Figure 7. Tuned TCgen specification

The CBP trace reader was written to favor compression ratio over decompression speed and was distributed without excessive tuning [25]. CBP uses a simpler set of predictors: gshare with 14 bits of history, a path-based indirect branch predictor with 2^{10} entries, a 128-entry return address stack (RAS), a static info cache with 2^{18} entries, and two target caches with a total of $2^8 + 2^{16}$ entries. Like VPC, a code is emitted which describes which predictors are correct. Unlike VPC, the code is followed by a variable-length record that contains only the information that must be corrected. CBP exploits the variable-length nature of x86 instructions. In addition, it supports all instructions, not just branches.

Though it uses similar techniques, a direct comparison with CBP is not possible (CBP obtains near-perfect program counter compression due to the interleaving of non-branch instructions). When perfect PC prediction is possible, CBP+bzip2 outperforms BPC in 10/20 cases, but when perfect prediction is not allowed, BPC produces smaller files.

In a sense, CBP does chaining as well but outputs the chain amount in a unary coding. For example, five 0's in a row means that internal predictors suffice to produce the next five branch records. With BPC, we merely output "5". While our encoding is simpler, the CBP encoding can lead to long runs of 0's that are easily compressed.

In conclusion, the specialized nature of our input data and our exploitation of long runs of correct predictions, allow for an extremely efficient implementation that generally exceeds the performance of more general related work.

3.5. Summary

Figure 8 summarizes the space and time information from Figure 4 and Table 4 and is a convenient way to choose the optimal compressor for a particular goal (speed or size) and dataset. For each of four workloads, we plot the average bits-per-branch and speed of decompressing for each class of traces. We use harmonic mean for the rate on the y-axis. The most desirable compressors, those that are fast and yield small file sizes, appear in the upper left of the plots. Note that gzip does not appear on the plot: it is the clear winner in speed, but its compression ratio makes it undesirable for snapshots as we saw in Figure 4.

For each application domain, BPC+PPMd performs the fastest. In terms of bits-per-branch, BPC+PPMd is similar to VPC for highly-compressible floating point traces and similar to PPMd for integer benchmarks. For multimedia, PPMd performs best, while BPC+PPMd performs significantly better than all its peers for hard-to-predict server benchmarks. High speed and small files across application domains are the strengths of our technique.

4. Related work

The similarity between data compression and branch predictors was noted in [7]. The authors reduced two-level direction predictors to implementations of Prediction by Partial Match (PPM) data compression [8] to show that two-level predictors were optimal in the asymptotic best-case. PPM has also been applied directly to direction prediction [10]; indirect branch target prediction [14]; and revisited in a recent submission to the Championship Branch Prediction (CBP) contest [17]. Despite asymptotic optimality, the resource-constrained CBP submission was outmatched by predictors that could better capture extremely long histories (80+ branches).

Branch trace compression has been attempted by reducing the entropy of a trace using blocking of symbols and employing short fixed-length codewords [19]. The dictionary and codewords are then passed to gzip for impressive compression ratios. This n-tuple scheme led to a more generic stream based compression scheme by the same authors [18] which has recently been surpassed by VPC [4]. Since BPC outperforms VPC for branch traces, we believe it would outperform the stream based compression as well, though different trace formats make a direct comparison difficult.

VPC exploits the equivalence between hardware predictors and data compressors to achieve excellent compression ratios on a wider spectrum of values. In fact, the generic value

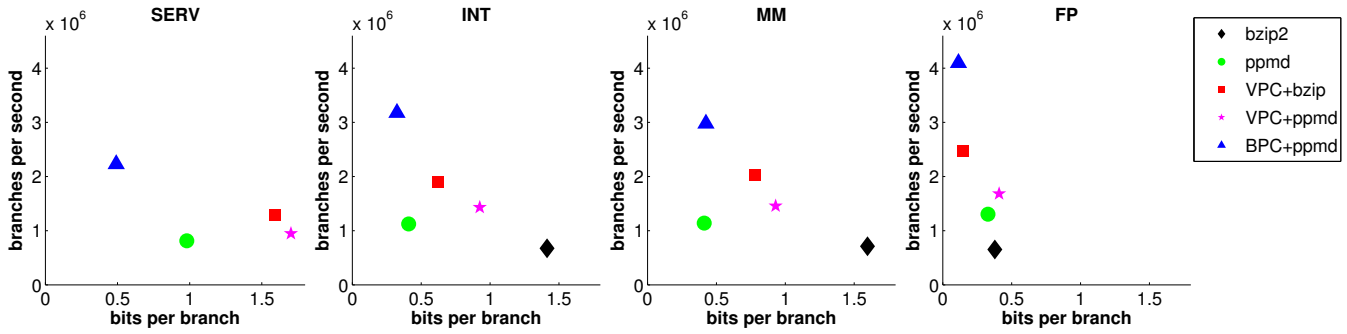

Figure 8. The optimal compressed trace format is in the upper left of each plot. Decompression speed across applications is reported with harmonic mean.

predictors used are quite useful for predicting indirect branch targets [5]. The BPC technique presented in this paper differs from VPC in two key ways. First, our branch traces are inherently more compressible than the extended data traces from the VPC work: instead of bzip2 ratios of 10–31X on average, we see 50–1000X. Second, VPC relies on multiple predictors in parallel, choosing the best-performing predictor for each record. BPC uses oracle information from a functional simulator to choose the single predictor most likely to be correct. This allows us to express correctness as a boolean, rather than a short code. Furthermore, it allows us to establish runs of correctly predicted branches and express these runs concisely.

The traces provided in the CBP contest are encoded using a public, but unpublished, scheme similar to VPC. They use a variety of predictors: a RAS, indirect BTB, and generic load value predictors to achieve compression [26]. As with BPC and VPC, the decompressor uses the same set of predictors, updating them along the way. When the compressor detects that many of its predictors are correct it need only send a short 1-byte code and the decompressor consults its copy of the predictors. Longer codes may be necessary to patch an incorrect prediction or provide unseen values. The CBP technique does not directly perform the chaining of BPC. While CBP is most similar to BPC, we were unable to perform a direct comparison because CBP relies on information from non-branch instructions to improve compression.

Memory Reference Reuse Latency is a different approach to speed warming [12]. Rather than store snapshots, an MRRL-enabled simulator runs in an online mode. To reduce prohibitive warmup times, only those instructions which occur within a statistically relevant near-history are used for warming prior to a detailed snapshot. Because the determination of relevance is based on memory interaction, subtle and long-lived branch correlations may cause inaccurate predictors, though in practice, there appears to be limited effect on estimated IPC. The accuracy may vary, however, when different branch predictor configurations are evaluated as MRRL metrics are based on load/store instructions instead of branches, and the correlation is not obvious. One could envision applying MRRL-like techniques along with BPC to limit the size of the trace to be compressed.

5. Conclusion

We have presented a technique, BPC, that utilizes software branch prediction structures to produce highly compressed branch traces for use in snapshot-based simulation. Utilizing fast, accurate predictors and chaining consecutive correct predictions, BPC achieves compression rates of 0.12–0.83 bits/branch (depending on workload), which is up to 210× better than gzip, up to 52× better than the best general-purpose compression techniques, and up to 4.4× better than recently published, more general, trace compression techniques. In the context of snapshot-based simulation, BPC-compressed traces serve as microarchitecture-independent representations of branch predictors. We have shown that this representation can require less space than one which stores just a single concrete predictor configuration, and that it permits the reconstruction of any sort of branch predictor.

6. Acknowledgments

We thank Joel Emer for suggesting the strategy; Martin Burtscher for discussions about VPC and the TCgen tool; and Jared Stark for releasing the CBP trace reader and explaining its motivation. Thanks also to the MIT SCALE group and anonymous reviewers. This work was partly funded by the DARPA HPCS/IBM PERCS project, NSF CAREER Award CCR-0093354, and an equipment grant from Intel Corp.

References

[1] K. C. Barr, H. Pan, M. Zhang, and K. Asanović. Accelerating a multiprocessor simulation with a memory timestamp record. In *Int'l Symp. on Perf. Analysis of Systems and Software*, Mar. 2005.

[2] M. V. Biesbrouck, L. Eeckhout, and B. Calder. Efficient sampling startup for sampled processor simulation. In *Int'l Conf. on High Performance Embedded Architectures and Compilers*, Nov. 2005.

[3] D. C. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[4] M. Burtscher, I. Ganusov, S. J. Jackson, J. Ke, P. Ratanaworabhan, and N. B. Sam. The VPC trace-compression algorithms. *IEEE Trans. on Computers*, 54(11), Nov. 2005.

[5] M. Burtscher and M. Jeeradit. Compressing extended program traces using value predictors. In *Int'l Conf. on Parallel Architectures and Compilation Techniques*, Sept. 2003.

[6] M. Burtscher and N. B. Sam. Automatic generation of high-performance trace compressors. In *Int'l Symp. on Code Generation and Optimization*, Mar. 2005.

[7] I.-C. K. Chen, J. T. Coffey, and T. N. Mudge. Analysis of branch prediction via data compression. In *Int'l Conf. on Architectural Support for Prog. Languages and Operating Systems*, 1996.

[8] J. G. Cleary and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Trans. on Communications*, 32(4), Apr. 1984.

[9] K. Driesen and U. Hölzle. The cascaded predictor: Economical and adaptive branch target prediction. In *Int'l Symp. on Microarchitecture*, Dec. 1998.

[10] E. Federovsk, M. Feder, and S. Weiss. Branch prediction based on universal data compression algorithms. In *Int'l Symp. on Computer Architecture*, June 1998.

[11] J. Gailly and M. Adler. gzip 1.3.3. `http://www.gzip.org`, 2002.

[12] J. W. Haskins and K. Skadron. Memory reference reuse latency: Accelerated warmup for sampled microarchitecture. In *Int'l Symp. on Perf. Analysis of Systems and Software*, Mar. 2003.

[13] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Trans. on Computers*, C-38(12), Dec 1989.

[14] J. Kalamatianos and D. R. Kaeli. Predicting indirect branches via data compression. In *Int'l Symp. on Microarchitecture*, 1998.

[15] R. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2), March-April 1999.

[16] A. KleinOsowski and D. J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, vol. 1, June 2002.

[17] P. Michaud. A PPM-like, tag-based predictor. In *Championship Branch Prediction Competition*, 2004.

[18] A. Milenkovic and M. Milenkovic. Exploiting streams in instruction and data address trace compression. In *IEEE Workshop on Workload Characterization*, Oct. 2003.

[19] A. Milenkovic, M. Milenkovic, and J. Kulick. N-tuple compression: A novel method for compression of branch instruction traces. In *Int'l Conf. on Parallel and Distributed Computing Systems*, Aug 2003.

[20] M. Milenkovic, A. Milenkovic, and J. Kulick. Demystifying intel branch predictors. In *Workshop on Duplicating, Deconstructing and Debunking*, May 2002.

[21] E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. In *Int'l Conf. on Parallel Architectures and Compilation Techniques*, Sep 2003.

[22] J. C. Ram Srinivasan and S. Cooper. Fast, accurate microarchitecture simulation using statistical phase detection. In *Int'l Symp. on Perf. Analysis of Systems and Software*, Mar 2005.

[23] J. Seward. bzip2 1.0.2. `http://www.bzip.org`, 2001.

[24] D. Shkarin. PPM: one step to practicality. In *Data Compression Conf.*, 2002.

[25] J. W. Stark. personal communication via email, Oct. 2005.

[26] J. W. Stark and C. Wilkerson et al. The 1st JILP championship branch prediction competition. In *Workshop at MICRO-37 and Journal of ILP*, Jan. 2005. `http://www.jilp.org/cbp/`.

[27] R. A. Sugummar. *Multi-Configuration Simulation Algorithms for the Evaluation of Computer Architecute Designs*. PhD thesis, University of Michigan, Aug. 1993. Technical Report CSE-TR-173-93.

[28] J. G. Thompson. *Efficient analysis of caching systems*. PhD thesis, University of California at Berkeley, 1987.

[29] T. F. Wenisch, R. E. Wunderlich, B. Falsafi, and J. C. Hoe. Turbosmarts: accurate microarchitecture simulation sampling in minutes. *SIGMETRICS Perform. Eval. Rev.*, 33(1), 2005.

[30] R. Wunderlich et al. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Int'l Symp. on Computer Architecture*, June 2003.