TCgen 2.0: A Tool to Automatically Generate Lossless Trace Compressors

Martin Burtscher
Computer Systems Laboratory, Cornell University
burtscher@csl.cornell.edu

Abstract

This tutorial explains the usage of TCgen, a tool for generating portable lossless trace compressors based on user-provided trace format descriptions. TCgen automatically translates these descriptions into optimized C source code. In many cases, the synthesized code is faster and compresses better than BZIP2, GZIP, MACHE, PDATS II, SBC, SEQUITUR, and VPC3, making it ideal for trace-based research and teaching environments as well as for trace archives. Version 2 includes several improvements and simplifies the integration of the generated code with other code through a streamlined API.

1. Introduction

TCgen is an application-specific compiler for translating user-provided trace descriptions into fast and effective trace compressors. A typical trace description comprises only a few lines of text (Section 3). TCgen emits portable C source code that is tailored to and optimized for the given trace specification. Compiling the resulting code will generate a standalone compression utility. Alternatively, users can integrate the code into other programs by replacing the `main` function. To facilitate this integration, the remaining functions adhere to a fixed API (Section 5). Moreover, the generated code is human readable to the extent that it is correctly indented, includes only one statement per line, and utilizes meaningful variable and function names. Users have a choice between several algorithms and combinations thereof to optimize the compression ratio and speed on their traces (Section 3.2).

TCgen employs value predictors [4], [6], [11], [12], [15], [16], [17] to convert a trace into streams that are highly compressible and that can be compressed and decompressed quickly with a general-purpose compressor. The following simplified example illustrates this process.

Assume we have selected a set of predictors and that we want to compress a trace containing records with a single field. During compression, the current field's value is compared with the predicted values. If at least one of the predictions is correct, the identification code of one of the correct predictors is written to the first stream. If all predictions are wrong, a reserved code is written to the first stream and the unpredictable value to the second stream. The predictors are then updated and the procedure repeats until all records have been processed.

Decompression proceeds analogously. First, one entry is read from the first stream. If it is the reserved code, the field's value is obtained from the second stream. If, on the other hand, the entry contains a predictor's identification code, the value from the corresponding predictor is used. The predictors are then updated to ensure that they make the same predictions as they did during compression. This process is iterated until the entire trace has been reconstructed.

While this method already compresses the traces somewhat, the key is that a general-purpose compressor is usually able to compress the resulting streams much faster and better than the original trace. TCgen users can select any compressor for this purpose that supports reading from the standard input and writing to the standard output.

A comparison with BZIP2 [7], GZIP [8], MACHE [14], PDATS II [9], SBC [13], SEQUITUR [10], and VPC3 [1] shows that a TCgen-generated compressor outperforms them on average and on the majority of the program execution traces we have tested in compression ratio, decompression speed, and compression speed [2]. The actual compression ratios we obtained range from 5.8 to 77161, with a geometric mean between about 40 and 800, depending on the type of trace.

TCgen can be accessed through a web portal at http://www.csl.cornell.edu/~burtscher/research/TCgen/. We have successfully tested a large number of compressors generated by TCgen on 64-bit UNIX systems as well as on 32-bit Windows machines.

The remainder of this tutorial is organized as follows. Section 2 introduces the value predictors available in TCgen. Section 3 presents the trace-specification language and sample specifications. Section 4 describes TCgen's code generation and optimization. Section 5 discusses the API of the generated code. Section 6 lists the differences between the previous and current version. Section 7 concludes the tutorial.

2. Value Predictors

This section describes the value predictors available in TCgen, i.e., the predictors the user can choose from and configure to optimize the performance of the compressor. Note that selecting predictors is optional. If none are selected, TCgen will use default predictors. All predictors predict the next trace entry based on previously seen entries. The parameters *x*, *n*, *t*, and *s* below are user definable.

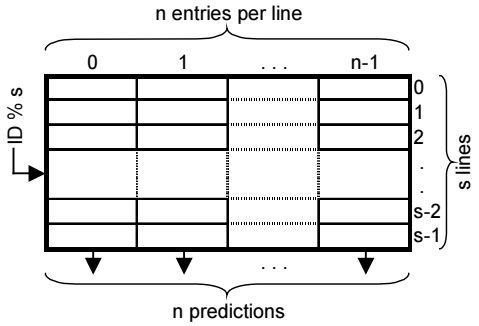

Figure 1: LV[*n*] predictor with *s* lines

Last-value predictor: The first type of predictor TCgen can emit is the last-value predictor LV[*n*] [4], [12], [17]. It predicts the *n* most recently seen values (with the same ID). This type of predictor can accurately predict sequences of repeating and alternating values as well as repeating sequences of no more than *n* arbitrary values. Figure 1 shows a diagram of an LV[*n*] predictor with *s* lines in its L1 table.

The LV[*n*] predictor always predicts the *n* values stored in the selected line. The ID (which is the value of one of the fields in the current trace record, see Section 3.2) modulo *s* determines the line index. If no ID is available, *s* has to be one. After a prediction, the selected line is updated by discarding the oldest entry, moving the remaining entries to the right by one slot, and copying the update value into the first slot. Note that the predictor is only updated if the first entry in the line is different from the update value.

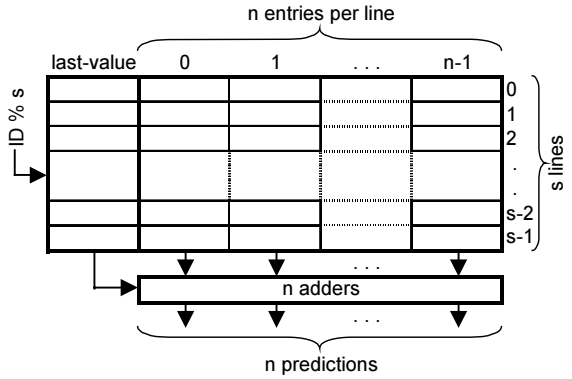

Figure 2: ST[*n*] predictor with *s* lines

Stride predictor: The second type of predictor TCgen can use is the stride predictor ST[*n*] [5]. It works just like the LV[*n*] predictor except it predicts and is updated with differences (strides) between consecutive trace entries rather than with absolute values. To form the final prediction, each predicted stride is added to the most recently seen value (the last value). This predictor can predict sequences of strided values, i.e., sequences with no more than *n* recurring differences between consecutive values. Figure 2 shows a diagram of an ST[*n*] predictor with *s* lines in its table.

The ST[*n*] predictor always makes *n* predictions, which are the *n* strides stored in the selected line to which the last value is added. The ID modulo *s* again determines the line index. If no ID is available, *s* has to be one. After a prediction, the selected line is updated by discarding the oldest entry, moving the remaining entries to the right by one slot, and copying the update stride and last value into the first and the last-value slot, respectively. The stride table is only updated if the first stride and the update stride differ.

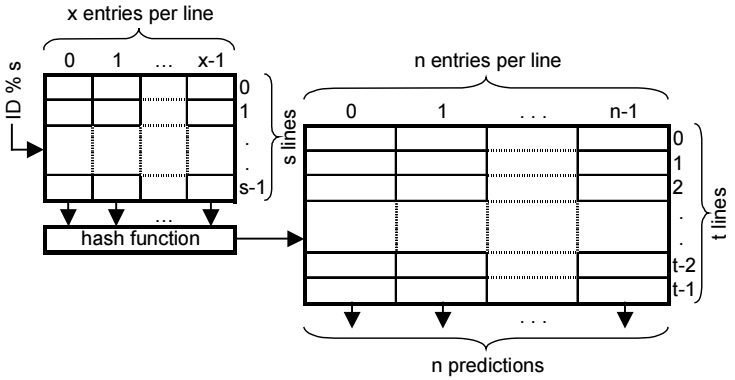

Figure 3: FCM*x*[*n*] predictor with L1 = *s* and L2 = *t*

Finite-context-method predictor: The third type of predictor TCgen can produce is the finite-context-method predictor FCM*x*[*n*] [16]. It computes a hash out of the *x* most recently encountered values (*x* is the *order* of the predictor), which are stored in the predictor's first-level table, using the select-fold-shift-xor hash function [15]. The hash is then used to index the predictor's second-level table (i.e., the hash table), which works just like the LV[*n*] table. After updating the second-level table, the entries in the selected line of the first-level table are moved to the right by one slot, thus dropping the oldest value and making room for the update value. Figure 3 shows an FCM*x*[*n*] predictor with *s* lines in the first-level table (L1) and *t* lines in the second-level table (L2).

This predictor predicts the values that followed the last *n* times the same *x* preceding values (that is, the same context) have been encountered [15], [16]. Thus, FCM predictors can memorize long arbitrary sequences of values and accurately predict them when they repeat.

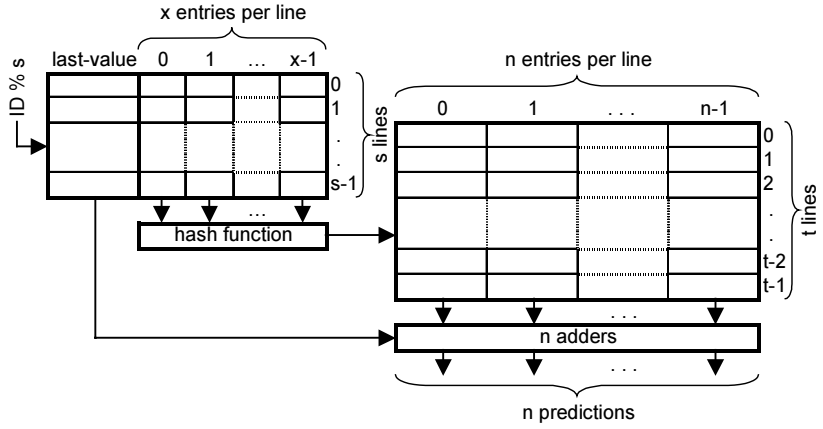

Figure 4: DFCM*x*[*n*] predictor with L1 = *s* and L2 = *t*

Differential-finite-context-method predictor: The fourth type of predictor TCgen can create is the differential-finite-context-method predictor DFCM*x*[*n*] [6]. It works just like an FCM*x*[*n*] predictor except it predicts and is updated with differences (strides) between consecutive trace entries rather than with absolute values. To form the final prediction, each predicted stride is added to the most recently seen value (the last value). DFCM predictors often outperform FCM predictors because they warm up faster, make better use of the hash table, and can predict values that have never been seen before. Figure 4 illustrates a DFCM*x*[*n*] predictor with *s* lines in the first-level table and *t* lines in the second-level table.

In addition to predicting long arbitrary sequences of values that repeat, DFCMs can accurately predict long arbitrary sequences of offsets (between consecutive values) that repeat.

3. Trace Specification Language

The input to TCgen is a trace format description combined with a value-predictor configuration and a second stage specification, expressed in the regular language whose grammar is shown in Figure 5. The start symbol is Description. TCgen also supports comments, which begin with a hash character ("#") and extend to the end of the line. The trace descriptions are case sensitive.

```
Description = "TCgen" "Trace" "Specification" ";" HeaderDef FieldDef {FieldDef} [IDDef] [CompressorDef] [DecompressorDef].
HeaderDef = Number "-" "Bit" "Header" ";".
FieldDef = Number "-" "Bit" "Field" Number ["=" "{" [LevelSizes] ":" Predictors "}"] ";".
LevelSizes = LevelSize ["," LevelSize].
LevelSize = ("L1" | "L2") "=" Number.
Predictors = Predictor {"," Predictor}.
Predictor = ("LV" "[" Number "]") | ("ST" "[" Number "]") | ("FCM" Number "[" Number "]") | ("DFCM" Number "[" Number "]").
IDDef = ("ID" | "PC") "=" "Field" Number ";".
CompressorDef = "Compressor" "=" "'" CommandLine "'" ";".
DecompressorDef = "Decompressor" "=" "'" CommandLine "'" ";".
Number = Digit {Digit}.
Digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
```

Figure 5: EBNF grammar of TCgen's description language

3.1 Trace Format Description

The following example highlights some of the features of TCgen's input language. Assume that our traces have no header and consist of a string of records, each with a four-byte and an eight-byte field. The corresponding trace format description is shown in Figure 6.

```
TCgen Trace Specification;
0-Bit Header;
32-Bit Field 1;
64-Bit Field 2;
```

Figure 6: Sample trace format description

All TCgen trace specifications have to start with the phrase "TCgen Trace Specification". A semicolon terminates each statement. The second line in Figure 6 informs TCgen that there is no header. Lines three and four specify that each record comprises two fields. Field 1 is four bytes and Field 2 eight bytes wide.

This specification suffices to generate a working compressor/decompressor for the given trace format. However, to improve the performance of the generated code, it is advisable to also specify (and experiment with) the predictors and their configuration.

3.2 Predictor Selection and Configuration

Figure 7 extends the above trace description with expressions that specify the types and sizes of the value predictors to be used for compressing the two fields. The last line tells TCgen to obtain the predictor IDs from the first field. Hence, no index is available for the first field and its level-one (L1) predictor size has to be one. Note that no matter where in a trace record the ID field is, it is always accessed first to make its value available for computing the indices to predict the remaining fields.

```
TCgen Trace Specification;
0-Bit Header;
32-Bit Field 1 = {L1 = 1, L2 = 131072: FCM3[2], FCM1[2]};
64-Bit Field 2 = {L1 = 65536, L2 = 131072: FCM1[2], ST[2], LV[4]};
ID = Field 1;
```

Figure 7: Sample trace format and predictor description

The specified FCM3[2] and FCM1[2] predictors provide four predictions for Field 1. For performance reasons, all FCMx and DFCMx predictors in the same field have a second-level table with $L2*2^{(x-1)}$ lines (see Section 4). Hence, the FCM1's hash table has 131,072 lines, i.e., the specified L2 value, but the FCM3's hash table actually has 524,288 lines.

The predictors for the second field have 65,536 lines in their first-level tables. There are three predictors, FCM1[2], ST[2], and LV[4], providing a total of eight predictions for Field 2. The last-value and the stride predictors do not have a second-level table, so the L2 value is irrelevant for these predictors.

Selecting predictors is optional. If no predictor is specified for a field, TCgen uses DFCM3[2], FCM3[2], and LV[2] by default. Providing L1 and L2 sizes is also optional, even if predictors are explicitly selected. If the sizes are provided, they have to be powers of two to make the index computations fast. If L1 is left out, a default value of one is used unless an earlier field with an L1 size of one already exists, in which case the default L1 value is 32,768. If L2 is omitted, it defaults to 65,536. If the optional ID specification is omitted, TCgen will select the first field with an L1 size of one. Note that it is always possible to compress traces without ID information by specifying an L1 size of one for all fields.

3.3 Second-Stage Specification

The generated code converts traces into streams, which should be further compressed with a general-purpose compressor (i.e., a second compression stage). If nothing is specified, TCgen compresses each stream by piping it into "*bzip2 -c -z -9*" and decompresses each stream using "*bzip2 -c -d*". These defaults can be overwritten, as illustrated in Figure 8. Note that the specified command lines have to invoke a compressor that is already installed on the target machine. Any compressor can be used that supports reading from the standard input and writing to the standard output.

```
TCgen Trace Specification;
0-Bit Header;
32-Bit Field 1 = {L1 = 1, L2 = 131072: FCM3[2], FCM1[2]};
64-Bit Field 2 = {L1 = 65536, L2 = 131072: FCM1[2], ST[2], LV[4]};
ID = Field 1;
Compressor = 'gzip -c -1';
Decompressor = 'gzip -c -d';
```

Figure 8: Sample trace format and predictor description with second-stage compressor specification

3.4 Usage

Entering a valid TCgen trace specification at http://www.csl.cornell.edu/~burtscher/research/TCgen/ will produce the desired C source code. The listing starts with a commented out copy of the trace specification to document the code. This specification is emitted in canonical form and includes a comment line after

each field specification stating how many predictions will be made for that field and what the size of the predictor tables is. The sum of the table size of all fields plus about 2MB reflects the dynamic memory requirement of the synthesized code.

When the generated code is compiled and run, it will compress traces that match the given specification. At the end of each compression run, predictor usage information is written to the standard output. This feedback is provided to help the user select the most effective predictors. For example, the output "605465 84.05% vpcF0fcm3[0]" states that the first prediction "[0]" of the FCM3 predictor "fcm3" for the first field "F0" was correct 84.05% of the time, i.e., it provided 605,465 correct predictions.

The generated source code (*vpc.c*) should be compiled into an executable (*vpc*) using a high optimization level. With *gcc*, the following command will generate the standalone compressor:

gcc -O3 vpc.c -o vpc

The executable compresses traces of the specified format from the standard input and writes two streams for each field into the current directory. For example, to compress the trace "*mytrace*", enter:

vpc < mytrace

Assuming the trace format from Figure 6, i.e., records with two fields, this will generate four files: *stream0a.vpc*, *stream0b.vpc*, *stream1a.vpc*, and *stream1b.vpc*. (We recommend concatenating and renaming the generated stream files using *tar* or some other suitable tool.)

To decompress the streams into the original file (*mytrace*), enter:

vpc -d > mytrace

Note that if you compress a file whose length minus the header size is not an integer multiple of the record size, the decompressed file will not have the correct length. Endian conversion is currently not supported, i.e., traces compressed on a little-endian machine can only be decompressed on little-endian machines. Only field sizes of one, two, four, and eight bytes are currently supported. However, the header can be any number of bytes long.

4. Code Generation and Optimization

When emitting code, TCgen employs several techniques to aid the compiler in producing a high-performance binary. For example, all code is written into one file (typically a few hundred lines of text), giving the compiler a global view of the program. All generated functions (except `main`) are declared static to allow the compiler to optimize the calling convention and to inline and eliminate any function it chooses. Similarly, all global variables are declared static. All local variables for which this is possible are declared as register variables to inform the compiler that no pointer analysis is necessary for them. Finally, all I/O is performed with efficient block I/O calls and the data is internally extracted from and inserted into buffers in a manner that avoids alignment problems.

TCgen performs several optimizations before emitting code, including dead code removal. For example, there is no need to compute a stride for a field that does not use a DFCM or an ST predictor. Similarly, if a trace format does not specify a header, no code to handle a header is emitted. TCgen minimizes the memory footprint of the generated code by eliminating unnecessary predictor tables, coalescing all tables that hold the same information, and minimizing the table sizes. For instance, all DFCM, ST, and LV predictors need to retain the last values. If only FCM predictors are specified for a given field, no last-value table is emitted. On the other hand, if multiple non-FCM predictors are specified, only one shared last-value table is generated. Furthermore, each FCMx and DFCMx predictor is assigned a second-level table with $L2*2^{(x-1)}$ lines. This is done to accelerate the computation of the hash function and so that only one first-level table is needed for all FCM predictors and only one for all DFCM predictors of each field. In fact, only the first-level table for the highest order predictor is generated and the lower-order predictors utilize whatever fraction of that table they need. All table elements are declared to be of the smallest type that is sufficiently large. Eight-bit fields, for instance, result in tables with eight-bit entries while 64-bit

fields result in 64-bit table entries. Likewise, elements of the smallest possible type are written to the streams that receive the unpredictable values. No matter which predictors are included, they are always "re-named" so that the predictor identification codes range from 0 to *n*-1, where *n* is the number of predictors. This improves the performance of the second stage compressor. TCgen eliminates superfluous parameters from functions that do not use them. For instance, the ID does not need to be passed to prediction functions for fields with an L1 size of one. Finally, TCgen code incrementally computes the hash functions, which results in substantial speedups for predictors with large orders. Moreover, the partial hash function values are chosen in such a way that they reflect the correct indices for the lower-order predictors (if they exist). As a result, only *n* operations have to be performed to compute the new index for an n^{th}-order FCM or DFCM predictor, and the intermediate results provide "free" indices for all lower-order predictors that are present.

5. Application Programming Interface

Users wishing to incorporate the synthesized code into their own programs should remove the `main` function, delete the `VPCread(buf, size)` and `VPCwrite(buf, size)` macros and implement them as real functions (see below), and call `VPCcompress()` and `VPCdecompress()` from appropriate places in their code. All other functions and global variables should be left unchanged. They are defined static and their names start with "VPC" or "vpc" to avoid conflicts with other code.

To simplify the code integration, TCgen explicitly defines the field types and the record size. For example, the trace description from Figure 6 results in the following definitions:

```
#define vpcrecordsize 12
#define vpcF0type signed int
#define vpcF1type signed long long
```

The two macros should be replaced by functions with the following prototypes:

```
size_t VPCread(void *buf, size_t size);
size_t VPCwrite(const void *buf, size_t size);
```

During compression, the synthesized code will call `VPCread` one or more times. Each time, the function should place up to `size` bytes of the data to be compressed into `buf` and return the actual number of bytes. During decompression, the synthesized code will call `VPCwrite` one or more times. This function should retrieve exactly `size` bytes of decompressed data from `buf` and return `size`. Any other return value will terminate the program.

6. Novelties in Version 2.0

The second version of TCgen presented in this tutorial includes several improvements over the previous version [3]. First, the predictor selection and the ID field definition are now optional. Second, we added the ST[n] predictor as well as the optional second-stage compressor/decompressor specification. Third, the dead code remover has been improved to also remove unused hash functions and variable definitions. Fourth, the generated listings are cleaner, all function and global variable names now start with "VPC" and "vpc", respectively, to avoid conflicts with other code, the record size is explicitly defined, and the VPCread and VPCwrite upcalls have been added to simplify the API. Fifth, the default L1 size has been modified to improve the performance. Also for performance reasons, the header is now included in the first stream instead of in a separate stream. Sixth, additional warning messages will be produced, such as if a predictor's hash table is not entirely reachable. Finally, we changed "PC" to "ID" to emphasize that the ID does not have to be a PC value. To maintain backwards compatibility, "PC" can still be used instead of "ID".

7. Conclusions

This tutorial describes the usage and features of TCgen, a tool that automatically synthesizes trace compressors from user-provided trace descriptions. While most other trace compressors in the literature have to be re-implemented every time the trace format changes, TCgen users merely have to provide a new format description, expressed in a simple specification language, and an optimized compressor will be generated in less than a second. Based on its flexibility, good performance, and portability, we believe TCgen to be a great tool for trace-based research and teaching environments as well as trace archives. It is freely accessible at http://www.csl.cornell.edu/~burtscher/research/TCgen/.

Acknowledgements

This work was supported in part by the National Science Foundation under Grant No. 0312966. We would like to thank Nana B. Sam for her help with TCgen 1.0.

References

[1] M. Burtscher. "VPC3: A Fast and Effective Trace-Compression Algorithm." *Joint International Conference on Measurement and Modeling of Computer Systems*, pp. 167-176. June 2004.

[2] M. Burtscher, I. Ganusov, S. J. Jackson, J. Ke, P. Ratanaworabhan, and N. B. Sam. "The VPC Trace-Compression Algorithms." *IEEE Transactions on Computers*, Vol. 54, No. 11, pp. 1329-1344. November 2005.

[3] M. Burtscher and N. B. Sam. "Automatic Generation of High-Performance Trace Compressors." *2005 International Symposium on Code Generation and Optimization*, pp. 229-240. March 2005.

[4] M. Burtscher and B. G. Zorn. "Exploring Last *n* Value Prediction." *International Conference on Parallel Architectures and Compilation Techniques*, pp. 66-76. October 1999.

[5] F. Gabbay. "Speculative Execution Based on Value Prediction." *EE Department Technical Report #1080, Technion - Israel Institute of Technology*. November 1996.

[6] B. Goeman, H. Vandierendonck, and K. Bosschere. "Differential FCM: Increasing Value Prediction Accuracy by Improving Table Usage Efficiency." *Seventh International Symposium on High Performance Computer Architecture*, pp. 207-216. January 2001.

[7] http://www.bzip.org/

[8] http://www.gzip.org/

[9] E. E. Johnson. "PDATS II: Improved Compression of Address Traces." *International Performance, Computing and Communications Conference*, pp. 72-78. February 1999.

[10] J. R. Larus. "Whole Program Paths." *Conference on Programming Language Design and Implementation*, pp. 259-269. May 1999.

[11] M. H. Lipasti and J. P. Shen. "Exceeding the Dataflow Limit via Value Prediction." *29th International Symposium on Microarchitecture*, pp. 226-237. December 1996.

[12] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. "Value Locality and Load Value Prediction." *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 138-147. October 1996.

[13] A. Milenkovic and M. Milenkovic. "Stream-Based Trace Compression." *Computer Architecture Letters*, Vol. 2, pp. 14-17. September 2003.

[14] A. D. Samples. "Mache: No-Loss Trace Compaction." *International Conference on Measurement and Modeling of Computer Systems*, Vol. 17, No. 1, pp. 89- 97. April 1989.

[15] Y. Sazeides and J. E. Smith. "Implementations of Context Based Value Predictors." *Technical Report ECE-97-8, University of Wisconsin-Madison*. December 1997.

[16] Y. Sazeides and J. E. Smith. "The Predictability of Data Values." *30th International Symposium on Microarchitecture*, pp. 248-258. December 1997.

[17] K. Wang and M. Franklin. "Highly Accurate Data Value Prediction using Hybrid Predictors." *30th International Symposium on Microarchitecture*, pp. 281-290. December 1997.