A Case for Compressing Traces with BDDs

Graham D. Price and Manish Vachharajani
Department of Electrical and Computer Engineering, University of Colorado at Boulder
Email: {Graham.Price,manishv}@colorado.edu

***Abstract*— Instruction-level traces are widely used for program and hardware analysis. However, program traces for just a few seconds of execution are enormous, up to several terabytes in size, uncompressed. Specialized compression can shrink traces to a few gigabytes, but trace analyzers typically stream the the *decompressed* trace through the analysis engine. Thus, the complexity of analysis depends on the decompressed trace size (even though the decompressed trace is never stored to disk). This makes many global or interactive analyses infeasible.**

This paper presents a method to compress program traces using binary decision diagrams (BDDs). BDDs intrinsically support operations common to many desirable program analyses and these analyses operate directly on the BDD. Thus, they are often polynomial in the size of the *compressed* representation. The paper presents mechanisms to represent a variety of trace data using BDDs and shows that BDDs can store, in 1 GB of RAM, the entire data-dependence graph of traces with over 1 billion instructions. This allows rapid computation of global analyses such as *heap-object* liveness and dynamic slicing.

I. Introduction

Performance tuning and hardware debugging make extensive use of instruction-level trace analysis [13]. Researchers have also suggested trace-based tools to debug complex software [14]. Unfortunately, faster processors and complex bugs increase the size of required traces. Trace files can easily grow to terabytes in size depending on the information collected and the duration of traced execution.

In response, researchers have developed streaming compression algorithms (e.g., Burtscher et al. [3]) that can compress these traces by a factor of 10 or more. Unfortunately, these techniques do not speed trace analysis. Compression techniques force analyzers to stream a decompressed version of the trace through the analysis engine. Thus, analyses have complexity that depends on the *decompressed* trace size, even though the full decompressed trace is never stored on disk. With large traces, this time-consuming process prohibits certain global analyses and interactive tools. Examples of prohibitively expensive operations include memory-data liveness visualization, trace slicing [15], and interactive visualization of application-level parallelism.

Ideally, a compressed trace format should allow analyses to operate directly on the compressed representation with complexity that is a function of *compressed* trace size. Then, if large portions of compressed traces fit in memory, global analyses and interactive visualization become possible. Larus et al. propose such a technique for *whole program path* analysis [7]. The technique uses the SEQUITUR compression method and works well for finding sequence matches in program execution. But, it does not permit direct application of data-centric analyses (e.g., trace slicing) that are of interest to system designers and programmers.

This paper presents a compression technique, based on Binary Decision Diagrams (BDDs) [2] that (1) supports data-flow centric analyses such as liveness and slicing, (2) has sufficient compression that large portions of program execution can be held in RAM, (3) allows efficient analysis, and (4) permits random access to trace data.

II. Traces as BDDs

Binary decision diagrams (BDDs) [2] are graph data structures that represent arbitrary boolean functions. Under the correct circumstances, they represent these functions compactly, achieving exponential reduction in size in many practical cases. Furthermore, operations used in trace analysis map to BDD operations. Worst-case, these operations complete in polynomial time in the number of nodes in the BDD, the size of the compressed representation [2]. Thus, BDDs are an attractive candidate for performing global and interactive analysis of large traces.

To use BDDs to compress and analyze traces, the following questions are answered below. (1) How does one represent a trace as a boolean function? (2) How does one analyze a boolean function representation of a trace? (3) How do BDDs compress the boolean function representation?

A. Traces as Boolean Functions

Boolean functions can be used to represent arbitrary sets of data if the possible elements of the set are encoded in binary. For example, if the universe (denoted Ω) has 4 elements α, β, γ, and δ, one can represent subsets of these elements by encoding them using two boolean variables x and y such that $\alpha = (0,0)$, $\beta = (0,1)$, $\gamma = (1,0)$, and $\delta = (1,1)$, where each tuple is of the form (x, y).

To represent the set $F = \{\alpha, \gamma\}$, one creates the boolean indicator function for F, $f(x, y)$, such that f evaluates to true if and only if its argument (x, y) encodes α or γ. In this case, $f = y'$ (i.e., not y). To test if some element λ is in a set with indicator function g, one evaluates g with arguments that correspond to λ. To see if α is in F, one computes $f(\alpha) = f(0,0) = 0' = 1$, and thus α is in the set F. $f(\beta) = f(0,1) = 1' = 0$, so β is not.

Using this idea, boolean functions, and thus BDDs, can be used to store arbitrary data. To represent the set of all program locations seen in a trace, one stores sets of n-bit unsigned integers, where n is the number of bits in the program counter (PC). Here the trivial binary encoding of each program

location is its PC value expressed in binary. Assuming a 4-bit PC encoded from MSb to LSb as (x, y, z, a), a trace that only executed the instructions at location 4 and 5 would be represented by the function $h = x' \wedge y \wedge z' = x'yz'$.

Tuples drawn from $\Gamma \times \Lambda$ can be used to represent the data addresses accessed by each program location, where Γ is the set of program locations and Λ is the set of possible data addresses. The fact that program location 7 accesses addresses 10, 11, 12, and 13, and that location 6 accesses addresses 10 and 11 is modeled by the set

$$F = \{(7, 10), (7, 11), (7, 12), (7, 13), (6, 10), (6, 11)\}$$

Now, let boolean vectors $\mathbf{g}$ and $\mathbf{l}$ indicate elements of Γ and Λ using the trivial encoding. If all memory locations are 4-bits ($||\mathbf{g}|| = ||\mathbf{l}|| = 4$) the indicator function for F, f, is

$$f(\mathbf{g}, \mathbf{l}) = g_3' g_2 g_1 l_3 l_2' l_1 \vee g_3' g_2 g_1 g_0 l_3 l_2 l_1'$$

where g_0 and l_0 are the LSb in the respective encodings.

The notion of sequence is the final element in understanding BDDs as trace representations. Instruction traces are usually stored as an ordered data structure. On disk the relative position of an instruction in a trace is determined by its relative position in the file. Older instructions are at earlier positions; newer instructions are at later positions. This notion of program order is critical for understanding dynamic properties such as instruction-level parallelism, data dependences, etc.

To capture the notion of order, assign every dynamic instruction in a trace a sequence number called the dynamic instruction number or DIN (e.g., the 1st instruction is 0, the 2nd is 1, the 100th is 99, etc.). Any pertinent data about an instruction trace is represented using sets of (DIN, data) tuples. For example, to store which instructions in the trace correspond to which program locations, one builds the set P that stores a (DIN, PC) tuple per instruction. To model the fact that the 45th instruction in the trace is at PC address `0x00ffede0`, one adds the tuple $(44, \texttt{0x00ffede0})$ to P. Thus P's indicator function p has $p(44, \texttt{0x00ffede0}) = 1$.

Note that it is often useful to store derived trace properties in BDD form. For example, many trace-based analyses, such as memory liveness or slicing, require the data-dependence graph (DDG) for all instructions in a trace (usually computed implicitly) [15]. The data dependence graph $G = (V, E)$ has as its vertex set, V, all the DINs in the trace. Thus, the data of interest is the edge set E, which is a set of (DIN,DIN) tuples. Note that since all address information is available during trace collection, the DDG can include dependences through memory as well as through registers.

B. Trace Analysis with BDDs

We now tackle the question of how to analyze traces stored using boolean equations (and thus BDDs). For brevity, a full treatment is omitted, but a method of doing forward slice analysis [15] using boolean equations is presented below. Slice analysis is the problem of finding the set of instructions that are part of a data-dependence chain that originates at (forward slicing) or ends at (backward slicing) a given set of instructions. Other work has described how to map common operations in other analyses to BDDs [12], [15], with Zhang et al. being the first to apply these ideas to bit-sets maintained during trace analysis.

function forward_slice(e,I_d)
  $s_{\text{old}} = 0$ // Empty set
  $s := I_d$
  do
    $e' := e \wedge s$
    $s := \exists \mathbf{d^1}.e'$
    $s = \text{rename}(\mathbf{d^2}, \mathbf{d^1}, s)$
  while($s \neq s_{\text{old}}$)
  return s

Fig. 1. Computing a Forward Slice using BDDs.

We have already described the simple analysis that tests for inclusion in some precomputed set. For example, to know if the instruction identified by DIN 1025 (from now on instructions will be referred to by their DIN number) is dependent on DIN 107, one can simply see if $(107, 1025)$ is in E (defined above). To do so, one computes $e(107, 1025)$, where e is E's indicator function. This operation is linear in the number of variables in the BDD for e [2].

Now, consider the problem of finding the set of all instructions that are directly dependent on an instruction with DIN d, in other words the set of immediate successors of d in the DDG (called an image computation [1]). To do this, one must extract all edges in E of the form (d, x), for arbitrary x. Denote this set by $\{(d, *)\}$. Assuming that d is encoded by $(d_0 = 0, d_1 = 0, d_2 = 1, d_3 = 1)$, the indicator function for $\{(d, *)\}$ is $I_{\{(d,*)\}} = d_0' d_1' d_2 d_3$. The set of all edges of the form (d, x) is then simply $E \cap \{(d, *)\}$. In terms of the indicator functions, one computes $e \wedge I_{\{(d,*)\}}$. Note that while we considered the case of direct dependents of a single instruction d, one could just have easily constructed a function I that represented all possible edges from a set of instructions and computed $e \wedge I$ to extract the edges. Note further that building BDDs for these functions is straight-forward and computing the logical and of two BDDs is polynomial in the BDD size [2].

Now all but one building block for the slice analysis algorithm are in place. Recall that one can extract the set of all tuples (d, x) that are in E. Call this set E'. One performs slice analysis by iterating this computation, but to do so one must extract the set of successor instructions $S = \{x | (z, x) \in E'\}$. One can then build the set $\{(S, *)\}$ and repeat the above computation until the slice is generated. To perform this extraction, one uses existential quantification (denoted $\exists$), an operation polynomial on BDDs. Given a function $f(x, \mathbf{y})$, where $\mathbf{y}$ is a vector of boolean variables and x a single boolean variable, $g = \exists x.f = f(1, \mathbf{y}) \vee f(0, \mathbf{y})$. To extract S defined above, one computes $S = \exists \mathbf{d^1}.E'$ where tuples in E' are of the form $(\mathbf{d^1}, \mathbf{d^2})$ and $\mathbf{d^1}, \mathbf{d^2}$ are boolean vectors.

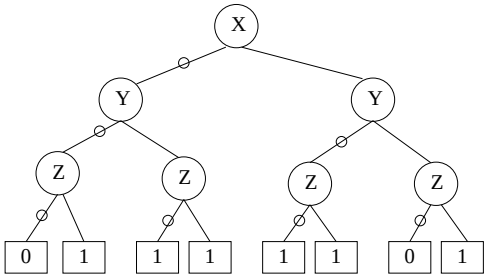
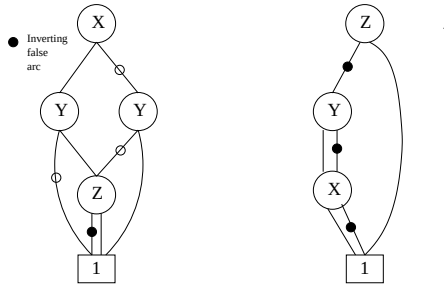


Fig. 2. A Three-variable Binary Decision Tree



(a) Naïve Order (b) Better Order

Fig. 3. A Three-variable ROBDD

Figure 1 shows how to compute the forward slice of an instruction with DIN d. In the figure, e is given as the indicator function for the set of edges in the data dependence graph, I_d is given as the indicator function for the instruction with DIN d, and s is the indicator function for the forward slice that is computed. The variables in indicator function I_d are in the vector $\mathbf{d^1}$, the variables in s are also in vector $\mathbf{d^1}$, and the variables in e are $(\mathbf{d^1}, \mathbf{d^2})$. The function rename takes a function s and renames the variables from set $\mathbf{d^2}$ to the corresponding variables in set $\mathbf{d^1}$. This operation is linear in the size of the BDD. Comparing two BDDs to see if they are equal is a constant time operation [2].

While the above describes a few simple analyses, BDDs can be used for much more sophisticated work. For a discussion, readers are referred to the references [1], [12], [15].

C. Compressibility of Trace BDDs

A boolean function can be represented using a binary decision tree. Figure 2 shows a binary decision tree of the three-variable function $f(x, y, z) = x'y + xy' + z$. Arcs with a circle correspond to when the variable is false, plain arcs to when the variable is true. Traversing the left edge of the graph, we find that $f(0, 0, 0) = 0$.

This tree can be compressed into a reduced ordered BDD (ROBDD) by repeatedly applying three rules. (1) Two nodes with the same label and children are merged. (2) Any node with identical children (and no inverting arcs, see rule 3) is removed and incoming arcs redirected to the child. (3) If a pair of nodes represent complementary functions (g and g'), the nodes are merged into one node for g, the arc to g' is redirected to g and marked as inverting. The '1' node is read as 0 if reached through an odd number of inverting arcs. Figure 3a shows the resulting BDD for f. In general, BDDs may compress the tree exponentially.

ROBDDs are canonical; equivalent functions have the same ROBDD. Note that inverting arcs allow equivalent functions to have different ROBDDs, but it is possible to always mark arcs as inverting in a consistent fashion to prevent this [8]. Thus, the compression the BDD achieves depends only on the function represented and the variable ordering. Compare the BDD in Figure 3a, which uses the order (X,Y,Z), to the smaller BDD for f in Figure 3b that uses order (Z,Y,X). Small changes in the ordering can result in order-of-magnitude changes in the size of the BDD. Note further that the best variable ordering depends on the function represented. For a detailed discussion of BDD variable ordering, readers are referred to the literature [4], [6], [9].

III. Results

Not all functions are amenable to BDD representation. This section shows that traces are amenable to BDDs by showing that trace data dependence graphs achieve compression ratios from 12x to 60x.

A. Experimental Setup

Experiments measure compression using BDDs to store the data-dependence graph (DDG) for true memory and register data-dependences (as described in Section II) for 64-bit PowerPC instruction-level traces of a number of gcc-compiled SPECint 2000 benchmarks. The BDDs themselves were generated and stored using CUDD [10] on a 32-bit platform. Each DIN in the tuple was a 64-bit number, for a total of 128 bits per tuple, and 128 variables in each BDD.

B. Compression Results

Figure 4 shows the number of tuples that fit into a fixed amount of memory. Note that as memory-size increases the number of stored DDG edges increases super-linearly in most cases. This is expected since BDDs can achieve exponential reduction in representation size [5], [11]. Though 181.mcf shows only linear growth, it achieves over 50x compression at all sizes shown (see below). Note further that in only 1 GB (2^{30} bytes) of RAM, we can store 500 million to 4 billion DDG edges for up to 1.04 billion instructions (12x to 60x more edges than the naïve representation). This clearly shows that BDDs provide the promise of large scale analysis within reasonable memory and time bounds.

Figure 5 shows how the compression ratio varies as the number of encoded operations increases. The compression ratio is the size of the naïve representation of the set of tuples (i.e., $16 \times$ number of tuples) divided by the size of the BDD (i.e., number of BDD nodes $\times$ BDD node size) . For clarity, only a representative set of benchmarks were included. All omitted benchmarks behave as scaled versions of 186.crafty. The scaling factor is roughly the ratio of the value of the 1GB bar for the omitted benchmark in Figure 4 to the value of the 1GB bar for 186.crafty. Benchmarks 181.mcf, 197.parser, and 175.vpr are outliers. Unlike the other benchmarks, 197.parser

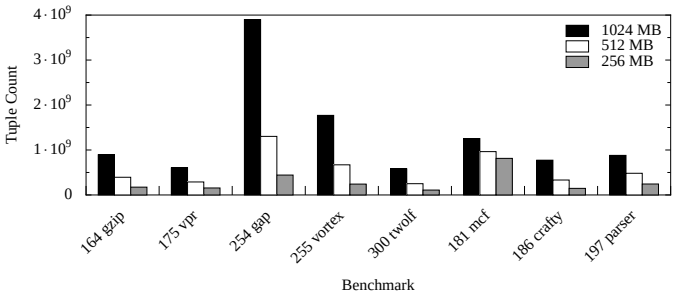


Fig. 4. Number of Tuples for Fixed Memory Bound

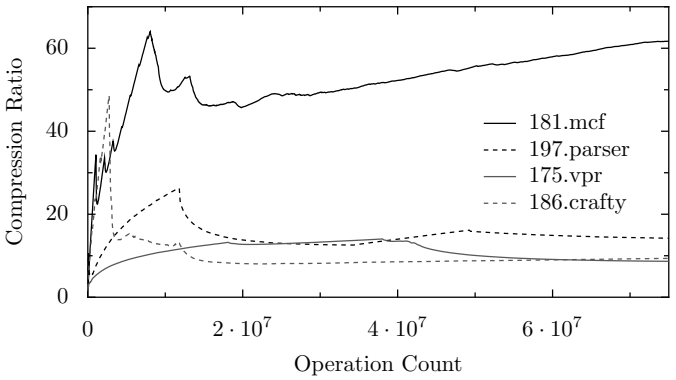


Fig. 5. Compression Ratio vs. Operations Processed

and 175.vpr show long-term decline in compression ratio after the initial spike. 181.mcf is an outlier because it initially shows excellent compression, only a modest dip, and then the usual long term increase. Also, note that some benchmarks show superb compression (e.g., 181.mcf) while most show modest ratios. Further study is needed to understand these variations.

To appreciate the performance of BDD-compressed trace analysis, consider the following. Using standard compression techniques [3], it takes 1.8s just to find the 2 millionth instruction in a trace of 186.crafty. Using BDDs, in one fourth the time, one can compute the 2 millionth instruction's forward slice to a depth of 1000 DDG edges.

IV. Conclusion

This paper shows that BDDs offer a method for efficiently storing and analyzing trace information. The method converts trace data into a boolean equation which is then represented using a BDD. The paper demonstrates that under the right variable ordering, the BDD representation offers excellent compression. Furthermore, unlike most prior compression techniques, analyses operate directly on the compressed representation (i.e., the BDD). Thus, analysis complexity depends on the compressed trace size. The paper demonstrates up to a 50x compression ratio when compressing PowerPC traces for SPECint 2000 benchmarks. With these compression ratios, 1 GB of RAM can hold the entire data-dependence graph of traces with over 1 billion instructions, allowing rapid computation of global analyses such as *heap-object* liveness, dynamic slicing, and many others.

Acknowledgments

The authors thank the reviewers for their insightful comments, Fabio Somenzi for his advice, Intel for support of this work, and Chinmay Ashok, Matthew Iyer, Josh Stone, and Neil Vachharajani for the Adamantium framework. Computer time was provided by NSF ARI Grant #CDA-9601817, NSF MRI Grant #CNS-0420873, NASA AIST grant #NAG2-1646, DOE SciDAC grant #DE-FG02-04ER63870, NSF sponsorship of the National Center for Atmospheric Research (NCAR), and a grant from the IBM SUR program. The ideas herein are not necessarily those of the above organizations.

References

[1] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. Vis: A system for verification and synthesis. In T. Henzinger and R. Alur, editors, *Eigth Conference on Computer Aided Verification (CAV'96)*, pages 428–432. Springer-Verlag, Rutgers University, 1996. LNCS 1102.

[2] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transaction on Computers*, C-35(8):677–691, August 1986.

[3] M. Burtscher and N. Sam. Automatic generation of high-performance trace compressors. In *Proceedings of the 2005 International Conference on Code Generation and Optimization*, pages 229–240, 2005.

[4] J. H. III and F. Brglez. Design of experiments in bdd variable ordering:lessons learned. In *Computer-Aided Design, 1998. ICCAD 98. Digest of Technical Papers. 1998 IEEE/ACM International Conference on*, pages 646–652, 1998.

[5] J. Jain, A. Narayan, C. Coelho, S. P. Khatri, A. L. Sangiovanni-Vincentelli, R. K. Brayton, and M. Fujita. Decomposition techniques for efficient robdd construction. In *Lecture Notes In Computer Science; Vol. 1166; Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, pages 419–434, 1996.

[6] S.-W. Jeong, B. Plessier, G. Hachtel, and F. Somenzi. Variable ordering for binary decision diagrams. In *Proceedings of the 3rd European Design Automation Conference*, pages 447–451, 1992.

[7] J. R. Larus. Whole program paths. In *Proceedings of the SIGPLAN '99 Conference on Programming Languages Design and Implementation (PLDI 99)*, pages 259–269, May 1999.

[8] J. C. Madre and J. P. Billon. Proving circuit correctness using formal comparison between expected and extracted behavior. In *Proceedings of the 25th Design Automation Conference (DAC)*, pages 205–210, 1988.

[9] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *ICCAD '93: Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 42–47, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[10] F. Somenzi. CUDD: Colorado University Decision Diagram package, release 2.30, 1998.

[11] F. Somenzi, K. Ravi, K. L. McMillan, and T. R. Shiple. Approximation and decomposition of binary decision diagrams. In *Annual ACM IEEE Design Automation Conference Proceedings of the 35th annual conference on Design automation*, pages 445–450, 1998.

[12] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 131–144, New York, NY, USA, 2004. ACM Press.

[13] Y. Wu and J. R. Larus. Static branch prediction and program profile analysis. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 1–11, December 1994.

[14] X. Zhang and R. Gupta. Whole execution traces. In *37th International Symposium on Microarchitecture (MICRO-37)*, pages 105–116, 2004.

[15] X. Zhang, R. Gupta, and Y. Zhang. Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams. In *26th International conference on Software Engineering (ICSE-26)*, pages 502–511, 2004.