

Compact Trace Trees in Dynamic Binary Translators

João Paulo Porto and Guido Araujo
IC - Unicamp
{jpporto, guido}@ic.unicamp.br

Youfeng Wu, Edson Borin and Cheng
Wang
Intel Corporation
{youfeng.wu, edson.borin,
cheng.wang}@intel.com

ABSTRACT

Trace Tree (TT) is a technique to collect program execution traces, which is commonly used in JIT environments. Its main features are the ability to perform loop unrolling and function inlining at no cost, while detecting application loop kernels. In this paper we evaluate a TT implementation in a DBT environment. We show that, under DBT, trace trees suffer from severe code duplication, considerably degrading its performance. In order to take advantage of the TTs interesting features in DBTs, we propose a variation called *Compact Trace Trees* (CTTs), which we show to be faster and to reduce code duplication.

Keywords

DBT, Trace Trees, Compact trace trees, hot spots

1. INTRODUCTION

Binary compatibility has been the standard for the microprocessor industry for several good reasons. First of all, maintaining binary compatibility implies the ability to run the existing (*legacy*) code. System programmers also benefit from backward compatibility, as previously acquired tool knowledge is still valid. For microprocessor architects, however, legacy compatibility also means that a lot of effort has to be put into newer chip versions, given that new products must run legacy code correctly, as if they were running in a previous chip set generation.

On the other hand, binary compatibility can also be implemented by using a special software layer. For example, in the Transmeta Crusoe[16] this is achieved by means of the *code morphing* layer. Intel has also explored software binary compatibility by using a tool called IA-32 EL[2], which allows x86 code to run on top of the Itanium processor.

As far as industry is concerned, dynamic binary translation, or simply DBT, is a viable way to keep existing code running on newer systems until new software comes up for the

new processors. As a drawback, DBT adds a considerable amount of overhead to legacy code execution. In order to amortize runtime overhead and to enable legacy code to run faster on newer machines, the runtime system must be able to perform code optimizations. Several techniques can be used, but knowing what to optimize is a key feature in any DBT. Hot trace detection is a set of algorithms that try to find these so called *hot regions*.

In order to achieve a low optimization overhead, while producing a decent runtime performance, hot traces must capture the kernel of the running applications. Tying more execution time to less code often means that translation and optimization will be much faster, thus allowing for even higher speedups.

Several techniques can be used to detect hot regions, varying according to the overhead they add to the running time. One well-known technique is MRET (*Most Recently Executed Tail* [1, 8]), which combines backward edges (and super blocks' side exits) to target profiling. MRET adds little overhead, when compared to the basic block profiling needed, for example, by the *Most Frequently Executed Tail*[7]). MRET fails, however, at detecting application's kernel, as the generated traces look like a dynamic CFG for the running application.

Franz et al[12] described a technique called *trace tree* (TT). Their experimental results show that TTs are capable of finding kernel-like traces. Also, the way trace trees are created, functions tend to be inlined, loops unrolled, or even inverted. The inner loop, where the program might spend most of its execution time, is likely to be discovered sooner by the runtime environment. Optimizing this early-discovered traces is an excellent way to hide optimization's overhead.

Unfortunately, trace trees can become potentially large due to its most notable characteristic: every path in a TT must end with a branch instruction to the entry point of the tree. To avoid this code explosion, long paths (i.e., paths longer than a maximum allowed threshold L) are discarded. Although it is possible to define L for a specific application, defining a general L requires a lot of effort. Also, defining L to be too small limits the effectiveness of TTs. In order to address these issues we propose at this paper the idea of *Compact Trace Trees* (CTTs), which is an extension of TTs which we show is less sensitive to L than TTs.

Algorithm 1: Trace Tree Creation rules

Input: BB : the next basic block to be executed by the program
Input: T : the TT being created or expanded

```
1 if  $BB = \text{anchor}(T)$  then
2   return Success
3 else if current edge is a back edge then
4   if there are too many back edges in  $T$  then
5     return Abort
6   else
7     increment back edge count in  $T$ 
8   endif
9 else if  $BB$  is root for any TT in the program then
10  return Abort
11 else if adding  $BB$  to  $T$  makes it too big then
12  return Abort
13 endif
14 add  $BB$  to  $T$ 
15 return Continue
```

This article is organized as follow: Sections 2 details the basic concepts behind TTs. Section 3 describes our implementation of the TT technique in a DBT runtime environment. Section 4 presents *Compact Trace Tree*, our *relaxed* approach to the TT technique. Section 5 presents the experimental results and a comparative analysis of TTs and CTTs. Previous and related work is discussed in section 6. Finally, section 7 presents the conclusions.

2. BACKGROUND ON TRACE TREES

Trace trees were first implemented on a Java Virtual Machine (*JVM*) for PowerPC [17], to support JIT compilation of an application most executed methods and loops. Algorithm 1 describes the rules for detecting trace trees. Before moving into the algorithm, some definitions are needed.

DEFINITION 1. *A trace tree T consists of several super blocks called tree-branches. The tree-branches are created as a set of the original program's basic block. The set of basic blocks in a Trace Tree T is denoted by $bbs(T)$.*

DEFINITION 2. *The first created tree-branch is called **root** of the TT, denoted by $root(T)$.*

DEFINITION 3. *The first basic block of the **root** of the TT is the **anchor**, denoted $anchor(T)$*

Algorithm 1, which is responsible for building the TT, is a very small automata, and is invoked whenever a branch instructions transfer the control flow from one basic block to another. It examines the current state (the parameter T) and decides whether it (1) continues the TT creation / expansion; (2) successfully terminates the process; or (3) aborts the process.

The first case happens when the algorithm reaches line 14. In that step, BB is added to T , and becomes part of the

Algorithm 2: Trace Tree link rules

Input: T : the TT being linked

```
1 foreach  $BB$  in  $bbs(T)$  do
2   if  $BB$  branches to  $anchor(T)$  then
3     link  $BB$  in  $T$  with the anchor
4   else if  $BB$  branches to tree-branch  $B$  of  $T$  then
5     //  $B$  was created to expand  $T$ 
6     link  $BB$  with  $B$ 
7   else
8     create side exit stub  $S$ 
9     link  $B$  with  $S$ 
10  endif
11 endfch
```

tree-branch being built. Line 15 indicates to the runtime environment that T is not yet completed, so the running program should continue to be executed in single-step (one basic block at a time) mode.

The second online case happens at line 2. If the process reaches that state, then the algorithm returns *Success* to the runtime environment to notify the successful creation or expansion of T , and that T is ready to be *linked*.

The last case (failure) happens when the algorithm reaches a state that makes it prohibitive to further record the trace. The algorithm then returns *Abort* to notify the runtime that T should be discarded. The issues mentioned in section 1 happens when the test on line 11 evaluates to true.

After a *Success*, T needs to be linked before it is ready to be used by the runtime system. During link-time, side exit stubs are created and unnecessary jumps are removed as shown in algorithm 2. The complete Trace Tree formation algorithm can be found in [12], and the interested reader is advised to read it for all the details.

2.1 Sample Trace Tree Creation

Consider the sample code in Figure 1(a). It scans a linked list in a loop, counting the number of zero elements, and invoking functions *func1* and *func2* (not shown in the example) for non-zero elements.

Considering the sample input in Figure 1(b), and assuming that the first element on the list is large enough to trigger the TT creation, a *tree-branch* is created, like the one shown in Figure 2(a). Notice that there is no back edge from block 15 to block 10, as, in TTs, the last instruction in a tree-branch jumps back to the first instruction in the tree, thus making back edges implicit.

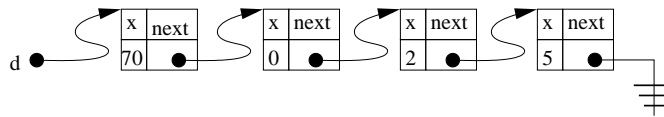
After the creation of this first tree-branch, program execution continues from the first instruction on it. However, this time, the side exit at node 10 is taken. What happens now is somewhat particular to TTs: the structure is expanded. If the new tree-branch conforms with the TT formation rules, then the newly created super block will be created as a *private* tree-branch for the tree, and will end with a branch to the first instruction in the TT. By doing so, the technique is capable of providing more detailed runtime information

```

(1) int func( data* d, int* sum ) {
(2)   unsigned int i;
(3)   int count = 0;
(4)   while ( d ) {
(5)     if ( (*d).x == 0 )
(6)       ++count;
(7)     else {
(8)       i = (*d).x;
(9)       do {
(10)        if ( i % 2 )
(11)          *sum = func1();
(12)        else
(13)          *sum = func2();
(14)        --i;
(15)      } while ( i > 0 );
(16)    }
(17)    d = (*d).next;
(18)  }
(19)  return count;
(20) }

```

(a) Sample C code



(b) The input data

Figure 1: Sample function

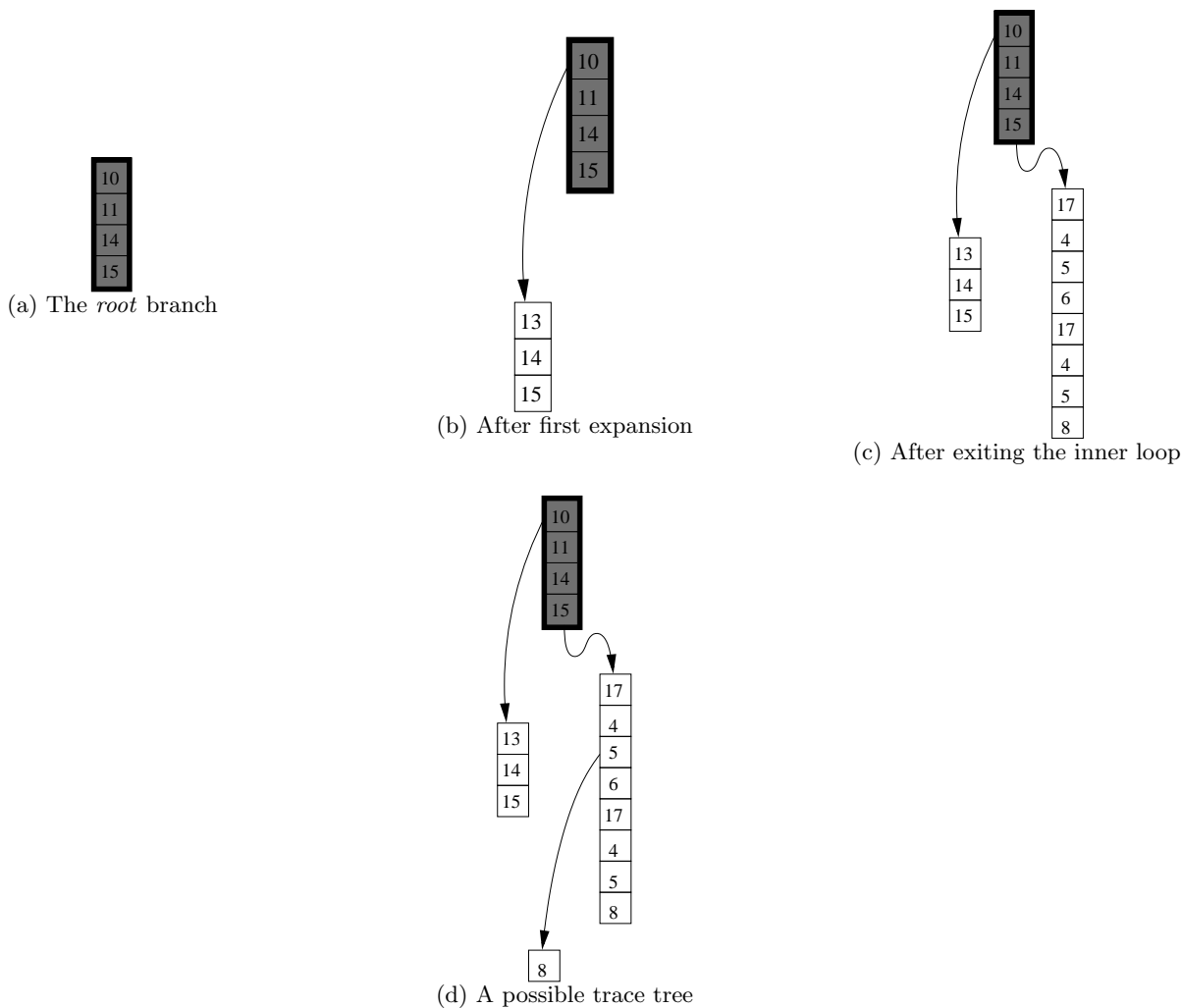


Figure 2: Sample Trace Tree creation

about the paths taken by the application. In the example, after the first expansion, the resulting tree can be seen in Figure 2(b).

After the inner loop finishes the execution, the side exit at node 15 is taken. When this happens, the trace expansion module starts to record the new tree-branch. After this expansion, the resulting trace can be found in Figure 2(c). Notice that nodes 17, 4 and 5 appear twice in this new tree-branch. This happens because of the “jump to the header” condition: every TT tree-branch **must** end with a jump back to the trace header. As a result, TTs might grow indefinitely, until a control transfer instruction jumps back to the header. Thus, some trimming rules are required to limit the TT growth, such as the maximum trace height, size or number of taken back edges.

Finally, the function will process the last node in the linked list, and the resulting tree in figure 2(d) results.

3. TRACE TREES ON DBT

One of the goals of this paper is to evaluate TTs using the SPEC2000 suite[21] running in a native execution environment. To do that, we implemented TTs in a DBT research framework, called StarDBT[22] developed by Intel. This framework is capable of running the whole SPEC2000 suite, as well as *regular* applications like Mozilla Firefox. The following sections gives an overview of our implementation, as well as describes some of our design decisions.

3.1 Detecting anchors

One interesting problem our implementation rose is how to detect *source file* anchors on an optimized binary. Since the compiler has the freedom to reorder basic blocks during code generation, we can not rely on backward branches to detect *source file* dominators. An obvious solution for this is to modify the compiler so basic blocks would not be reordered. Although this is a feasible, it is not practical, and could mask the results, as we intend to understand the technique behavior on native, fully optimized binaries.

Yet another solution would be to keep an array with the recently executed basic blocks. This solution would work fine if it did not raise two issues. First, the array needs to be traversed every time a control transfer instruction is executed. As the average basic block is small, the array would be scanned every five or so instructions, considerably degrading performance. The second issue is the array length. We have no means to predict how many basic blocks we need to keep in order to detect the loops. If the array is not big enough, and is used in a circular fashion, no anchor will eventually be found.

After evaluating these possibilities, we decided to use the backward branch heuristic. If a backward branch occurs in the compiled program, then it is likely to be part of a loop in the original source. Either that, or the profiled edge is not part of a loop, in which case it is unlikely that it would trigger a trace creation. Our experimental observations show that this approach works pretty well in practice.

3.2 Indirect branches

On IA-32 architecture, indirect branches are hard to handle due to the processor’s byte addressable memory, variable instruction size and no alignment requirement for the instructions. In other words, indirect jumps can jump anywhere into the processor’s address space, so care must be taken when handling indirect branches on DBT systems.

Had the IA-32 ISA alignment restrictions or only fixed-size instructions, a table-based address translation could be used (as in [23]) to perform instruction address lookup on traced’ indirect branches. However, this table would be considerably large, possibly requiring all (or even more) memory the computer can address.

To overcome this problem, our implementation keeps a list of possible targets for each indirect branch in the trace which was executed at least once. Indirect branches are generally not very common in the SPEC2000, with the exception of GCC and C++ programs, as indirect branches are used to implement virtual function calls. Thus our experiments did not show a severe impact on the translated binary performance. Needless to say, memory requirements were much lower and, most importantly, could be met.

Several techniques for handling indirect branches are described by Hisser et al. [15]. They perform several experiments regarding indirect branches and provide experimental evidence that there is no best technique. In our system, we use a local version of the *Indirect Branch Translation Cache* technique.

3.3 Trace expansion

When dealing with the expansion of the trace, the variable size instruction on the IA-32 posed another obstacle to our implementation.

Every time a trace tree is linked, its side exits are compiled to branch to expansion stubs. These stubs set up data needed for the trace expansion before switching to the runtime environment that will actually start the expansion. Example of such data are pointers to the current executing trace, that are used to patch after expansion.

To overcome this, we forced branches on side exit to use the 32-bit branch instruction format. These instructions are generally 6 bytes long, as opposed to the 2-byte short format. By using the long format, we added some overhead to the processor’s front-end (*decoding unit*), as well as increased L1 cache usage. We could not identify other simpler way to implement this feature and, since trace expansion is key to the TT technique, long branches were used.

3.4 Code duplication due to Path Specialization

The way TTs were designed, every time a side-exit is expanded, a new tree-branch is generated. This new tree-branch can be thought as an specialization since there will be no join points (as shown in figure 2(d)). In other words, tail duplication might lead to severe code duplication.

Unfortunately, as discussed in our experimental results (section 5.1), duplication was somewhat high. In order to ad-

Algorithm 3: Compact Trace Tree Creation rules

Input: BB: the next basic block to be executed by the program
Input: B: the new tree-branch being created
Input: T: the CTT being created or expanded

- 1 **if** $BB \in bbs(B)$ **then**
- 2 add B to T
- 3 **return** *Success*
- 4 **else if** $BB \in anchor(path(B))$ **then**
- 5 add B to T
- 6 **return** *Success*
- 7 **else if** BB is root for any TT in the program **then**
- 8 add B to T
- 9 **return** *Success*
- 10 **else if** adding BB to T makes it too big **then**
- 11 discard B
- 12 **return** *Abort*
- 13 **else**
- 14 add BB to B
- 15 **return** *Continue*
- 16 **endif**

dress this issue, we could either choose to have a smaller dynamic coverage to keep the duplication (memory usage) low, or have a high duplication level, thus increasing the coverage.

Tail duplication is an important transformation that enables some optimizations and may expose parallelism[5, 19]. In order to still maintain tail duplication, and to avoid some of the duplication, we propose a modified, less strict set of trace formation rules for TTs. We named this relaxed form *Compact Trace Trees* (CTTs).

4. COMPACT TRACE TREES ON DBT

Compact trace trees were named *compact* as it is an attempt to have some interesting features of TTs, while keeping duplication low. Before explaining our TT modification, some definitions from TTs need to be relaxed.

DEFINITION 1. *A CTT consists of several super blocks called tree-branches. The tree-branches are created as a set of the original program's basic blocks. The set of basic blocks in a tree-branch B is denoted by $bbs(B)$.*

DEFINITION 2. *The first created tree-branch is called root of the CTT.*

DEFINITION 3. *Every tree-branch's first basic block is an anchor. For a tree-branch B , $anchor(B)$ denotes its anchor.*

DEFINITION 4. *Path is the tree-branch sequence from the current tree-branch B to the root, denoted by $path(B)$.*

The rules to create the CTTs based on the relaxed definition of TTs is shown in Algorithm 3. When comparing it to

Algorithm 4: Compact Trace Tree link rules

Input: T: the CTT being linked

- 1 **foreach** $B \in T$ **do**
- 2 **foreach** $BB \in B$ **do**
- 3 **if** BB branches to an A anchor($path(B)$) **then**
- 4 link BB with A
- 5 **else if** BB branches to any CTT C **then**
- 6 link BB with C
- 7 **else**
- 8 create side exit stub S
- 9 link BB with S
- 10 **endif**
- 11 **endfch**
- 12 **endfch**

the Algorithm 1, one can notice that there is now only one condition that might make the trace creation/expansion to abort. The relaxed *conditions* enable the implementation of a relaxed *automata*.

The *link-time* part of the implementation (algorithm 4) needed some small changes to work. Line 3 needs to scan through a list of tree-branches to search for the successor of all the branches it links.

Another interesting feature is found on line 6, where one CTT is linked with another CTT. Notice that this link only allows one CTT to jump to the *entry* of another CTT.

The modifications we implemented make impossible to infer back edges for the trees. In fact, a tree-branch might even end with a branch to a *cold-block*. This might happen when the algorithm returns *Success* on line 3 of algorithm 3. Because of this feature, all control flow edges must be explicitly represented in CTTs, as seen in Figure 3.

4.1 Compact Trace Tree creation example

The first two CTT tree-branches are created the very same way as previously seen on Figures 2(a) and 2(b), so this step will not be shown this time. Figure 3(a) shows the initial CTT right before the side exit at node 15 is taken. The Figure is presented here to show the (now) non-implicit back edges.

When node 15's side exit is taken, the trace expansion module starts recording a new branch for the CTT. This time, however, when the application jumps back to instruction 17, the trace expansion module will detect this as a trace recording stop condition, and will stop recording the branch. During link time, the jump to node 17 will be treated as a jump to an anchor, and the resulting CTT is shown in Figure 3(b).

Now, when instruction 5 takes the side exit to instruction 8, trace expansion is triggered again. The instruction that follows 8 is 10, which is the anchor of the root branch. This way, expansion will be completed and the resulting CTT is shown in Figure 3(c).

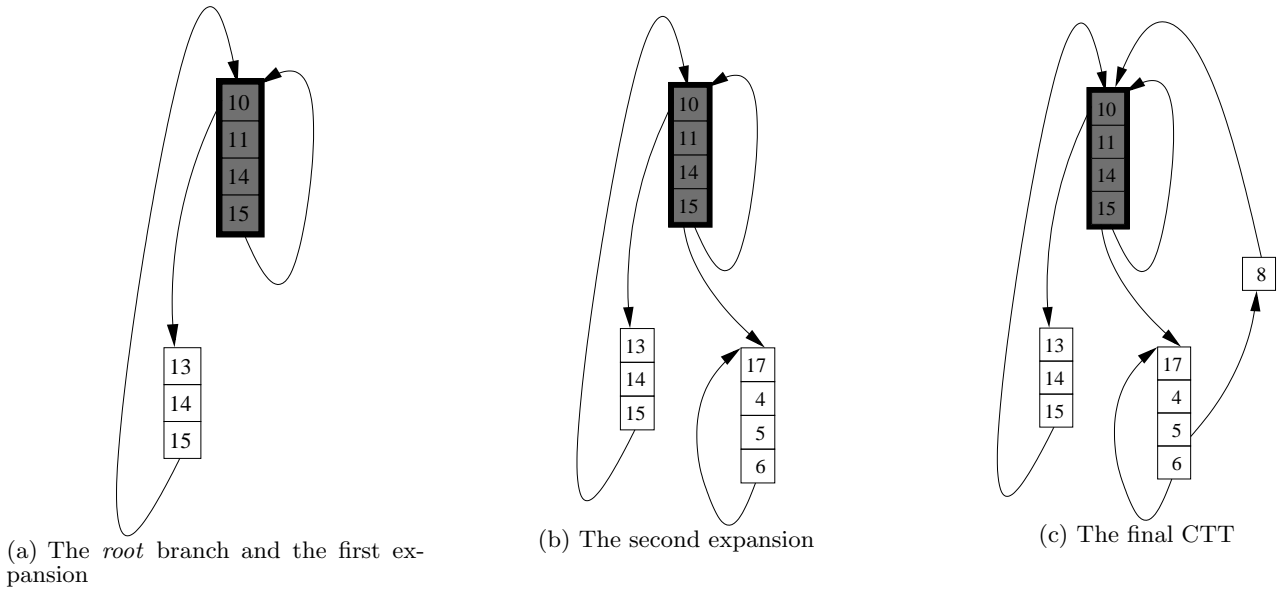


Figure 3: Sample Compact Trace Tree creation

5. EXPERIMENTAL RESULTS

In order to compare TTs and CTTs, we ran experiments using both techniques and different maximum tree heights values, where *tree height* is defined as the basic block count from the *root's anchor* of TT or CTT up to the farthest block from it.

Also, we ran experiments with the MRET trace formation rules. For MRET traces, the maximum tree height served as a superblock size limit. Since MRET traces are not expanded, this parameter would have no other logical meaning.

The experiments were ran under Ubuntu 8.04.1 running on an Intel Q6600 2.4 GHz, with 4GB of RAM 1066MHz and two 500GB SATA disks. We used the 32 bit Linux distribution since our DBT is targeted at IA32. The whole SPEC2000 suite was used, under the reference input, and all the binaries were statically compiled, with optimization level 2.

The parameter (tree height) ranged from twenty blocks to a hundred and twenty, in twenty blocks steps. We also ran a special experiment with maximum tree height of five blocks. Table 1 shows how dynamic coverage changes with this parameter. It also shows that TTs and CTTs are sensitive to it. As expected, the greater the trace height, the better the coverage.

Interesting enough, notice that MRET traces are not sensitive to the height. MRET gets the higher coverage numbers of all techniques no matter the tree height is used (or block count for MRET super blocks). As anticipated, TT shows the expected behavior of increasing coverage as tree height becomes higher. It also exhibits an asymptotic behavior around 80%. For CTT, coverage seems slightly better, as the asymptotic behavior shows up around 86%.

Time spent within the runtime is shown in table 2. Again,

MRET shows no sensitivity with respect to the trace height. CTT seems to be stabilizing as the maximum trace height is increased, which is good, since a large height would not affect the technique much. TT however, seems to be suffering with the higher trees. A quick notice, regarding the last two values for CTT on this table: they are smaller than the others since the last instance of bzip2 could not finish. Had it finished, its runtime would be higher.

Table 3 shows the duplication numbers for our instances. As shown, our technique fulfilled our primary goal to decrease duplication. Again, the last two entries for TT lack the last bzip2 instances, being slightly different than what was expected.

Finally, table 4 indicates the memory usage for a complete run of the SPEC2000 suite. It is noticeable how memory usage increases for CTT and TT, as a higher trace height was used. Again, the MRET implementation is independent of the parameter.

5.1 TT on DBT

Trace trees did not perform as well in DBT as it did on the Java Virtual Machine. The technique seems to be really dependent upon a good estimate of trace height to achieve good results. Selecting a small trace height value is bad, since it might lead the technique to under-perform. For example, in Figure 4(a) the technique performed well until the last three experiments. The only parameter that changed was the maximum trace height.

In the same benchmark, as shown Figure 4(b), duplication became an issue from the fourth experiment on, and, if duplication rises, so does memory usage. Notice the poor performance on Figure 4(a). Since duplication increases exponentially, the runtime system takes much longer to create the TT.

Block count	CTT(%)	MRET(%)	TT(%)
5	50.53	97.70	48.79
20	74.72	97.59	67.42
40	81.97	97.62	74.32
60	82.78	97.63	76.13
80	85.05	97.60	76.37
100	85.56	97.60	76.94
120	85.64	97.60	77.88

Table 1: Average coverage for all experiments

Block count	CTT	MRET	TT
5	105	68	97
20	96	68	96
40	117	68	115
60	137	68	162
80	137	68	264
100	141	68	225
120	145	68	242

Table 2: Total time on the framework (minutes)

The bzip2 benchmark also suffered with bad performance. As shown in Figure 5, TT was not capable of finding loop kernels, and this led to the code explosion seen in Figure 5(b).

Also, the technique suffered from the highest duplication (see table 3). Specifically bzip2 had problems with the last two instances of the last two experiments: system memory was exhausted. Since each experiment is composed of 3 instances, we only show the data for the instance that finished the experiments.

The problems faced here by TT do not invalidate it as a trace technique. They just show some aspects that need to be addressed by system writers if they want to use TT.

5.2 CTT on DBT

CTT successfully addresses the issues found with TT in our system. As tables 1, 2 and 3 show, CTT’s coverage was higher than TT’s, duplication was smaller, as was execution time. It is interesting to notice that CTTs are less sensitive to a poor trace height parameter, at least with respect to the execution time. Table 4 shows that CTT consumed less memory than TT.

Figure 4 shows that, with a high enough trace height parameter, CTT can behave much like MRET. Particularly, Figure 4(a) shows that CTT was as fast as MRET. However, Figure 5(b) shows that CTT can be very different from MRET, regarding duplication. Overall, the technique is competitive (figure 5(a)) with MRET.

5.3 MRET, TT or CTT?

The three trace techniques are valid trace techniques that have their own advantages as well as disadvantages. MRET is, by far, the fastest algorithm. Also, as expected, it is not sensitive to trace size parameter, making it the ideal choice when there is no available time for calibration. Unfortunately, it lacks tail duplication, and runtime analysis of the

Block count	CTT	MRET	TT
5	0.08	0.42	0.06
20	0.62	0.52	0.33
40	2.34	0.54	5.83
60	4.43	0.54	28.36
80	5.37	0.54	73.75
100	6.75	0.54	69.21
120	6.85	0.54	76.19

Table 3: Average duplication

Block count	CTT	MRET	TT
5	939	980	939
20	989	989	963
40	1130	990	1205
60	1311	990	2105
80	1402	990	3498
100	1535	990	3407
120	1549	990	3700

Table 4: Total memory usage (in MB)

generated dynamic CFG might be slow.

TTs tend to perform poorly on DBT systems. We observed that the technique is slow and is not capable of detecting loop kernels (or even loops) in some applications. This is unexpected as the previous results[12] were good, and might be explained by the fact that an IA-32 DBT presents a much more challenging environment than a Java Virtual Machine. TTs also suffers a performance hit when the tree height is too big.

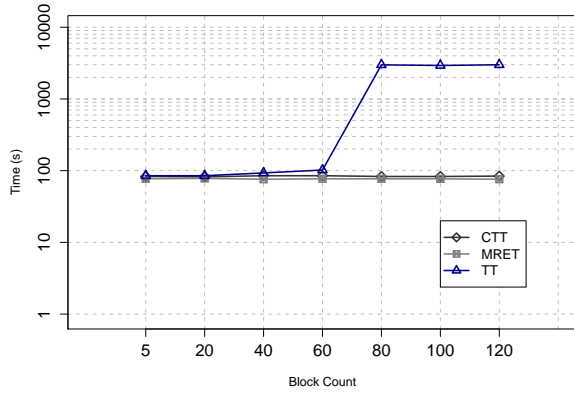
CTTs is a good choice for DBTs. It provides an efficient algorithm runtime with good coverage results, while keeping duplication at reasonable levels. Our experiments show that CTTs will still perform well even if the selected tree height is larger than necessary, thus eliminating the need to find its best value.

5.4 Interesting results

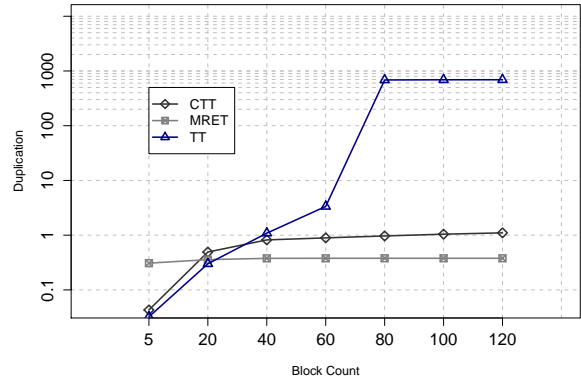
Although all the techniques behaved consistently, some interesting results were found. Two interesting results were selected and are show in Figure 6. They are briefly discussed below.

Figure 6(a) shows that coverage may drop even if we increase the maximum trace height. This result is counter-intuitive, since it is expected that, as the maximum allowed trace height is increased, coverage increases as well. This may be happening because, with a higher tree, the runtime environment keeps trying to create a long tree branch (which ends up being aborted), while the runtime with the more strict rules gives up on the long trace and end up detecting a smaller, more important trace. It is curious to notice that both CTT and TT suffered the same problem, but at different tree height values.

Figure 6(b) shows two other interesting aspects. First, it looks like CTT coverage would still increase as trace height increases. In other words, it seems that the loops in vortex

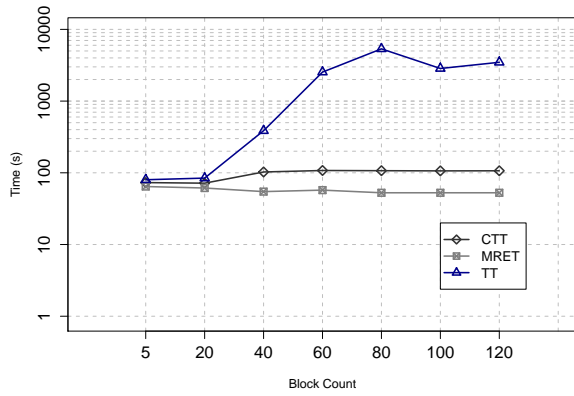


(a) Time

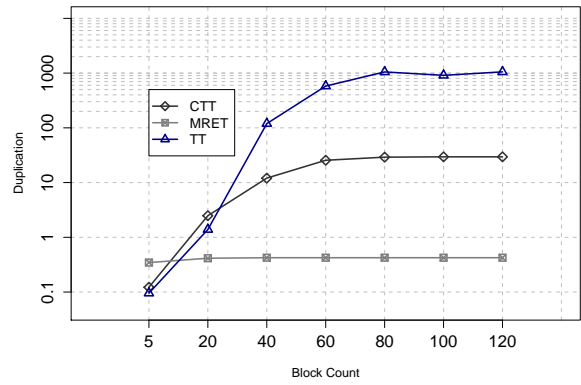


(b) Duplication

Figure 4: 181.mcf – SPEC2000 int

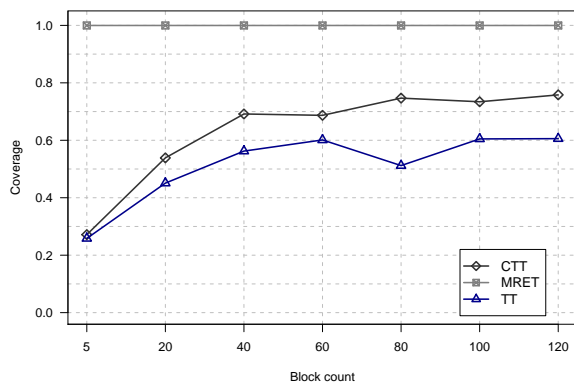


(a) Time

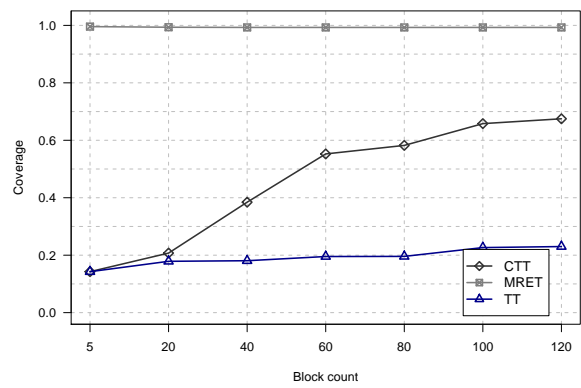


(b) Duplication

Figure 5: 256.bzip2 – SPEC2000 int



(a) 197.parser – SPEC2000 int



(b) 255.vortex – SPEC2000 int

Figure 6: Interesting results

are big. In other words, CTT needs a big tree height to be able to capture the whole loops.

Regarding the same benchmark, it is interesting to notice that, albeit being very similar when compared to CTT, TT coverage did not increase much with the parameter change. Here, the very restrictive *branch to the anchor rule* (line 7 of algorithm 1) prevent the algorithm to find the loops.

Also, for both benchmarks shown in Figure 6, MRET performs perfectly, as it does on most of benchmarks from SPEC200 suite (both floating point and integer benchmarks). As a result MRET can be used as an upper bound on the maximum coverage that can be obtained by CTT or TT.

6. RELATED AND PREVIOUS WORK

Dynamic binary translation is used in DAISY [9, 10, 11, 13] to achieve compatibility between two different architectures. The work is not tied to any particular architecture, thus making the techniques suitable for almost any ISA available. Unlike DAISY, our system translates IA-32 to IA-32. Also, the authors describe a page-based translation scheme, whereas our system operates on classical basic blocks that have no page alignment requirements.

The effects of tail duplication on hyperblock formation can be found on [18, 19]. In addition to tail duplication, [18] performs *head* duplication to perform convergent hyperblock formation.

Traces can be used to speed up system simulators (as described in [20]). Traces are also good units for parallelization [3, 4].

Prior to Trace Trees, Havanki et al [14] used tree regions code generation units. However, the *treeregion scheduling* was statically performed to generate code for wide issue processors, whereas Trace Trees (and Compact Trace Trees) use a dynamic approach to generate the trees.

Our framework use software profiling without hardware support to collect hot regions. Chen et al. [6] describe hardware sampling to collect traces with a lower overhead.

7. CONCLUSIONS

This paper presents an evaluation of the Trace Tree technique running in a IA32 DBT. It presents experimental evidences that the technique, as described previously, is not suited for the challenges presented by both DBT and IA32.

This paper also presents a novel technique for trace construction that seems to perform well under a DBT, being a middle ground between traditional MRET traces and Trace Trees.

We also present experimental evidences that MRET is a good choice when no tail duplication is needed, and when runtime performance must be high and memory usage should be kept low. Also, MRET seems to be an upper bound estimate on trace techniques behavior.

8. ACKNOWLEDGMENTS

This work was partially sponsored by CNPq, CAPES and Intel Corporation. We would like to thank Dr. Mauricio Breternitz Jr. for bringing TTs to our attention. We also would like to thank the reviewers who helped improving this paper with invaluable suggestions.

9. REFERENCES

- [1] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *SIGPLAN Not.*, 35(5):1–12, 2000.
- [2] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skalesky, Y. Wang, and Y. Zemach. IA-32 execution layer: A two phase dynamic translator designed to support IA-32 applications on Itanium®-based systems. In *36th International Symp. on Microarchitecture*, pages 191–202, 2003.
- [3] B. J. Bradel and T. S. Abdelrahman. The potential of trace-level parallelism in java programs. In *PPPJ '07: Proceedings of the 5th international symposium on Principles and practice of programming in Java*, pages 167–174, New York, NY, USA, 2007. ACM.
- [4] B. J. Bradel and T. S. Abdelrahman. A study of potential parallelism among traces in java programs. *Sci. Comput. Program.*, 74(5-6):296–313, 2009.
- [5] P. P. Chang, S. A. Mahlke, and W.-m. W. Hwu. Using profile information to assist classic code optimizations. *Softw. Pract. Exper.*, 21(12):1301–1321, 1991.
- [6] H. Chen, W.-C. Hsu, J. Lu, P.-C. Yew, and D.-Y. Chen. Dynamic trace selection using performance monitoring hardware sampling. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 79–90, Washington, DC, USA, 2003. IEEE Computer Society.
- [7] C. Cifuentes and M. V. Emmerik. Uqbt: Adaptable binary translation at low cost. *IEEE Computer*, pages 60 – 66, March 2000.
- [8] E. Duesterwald and V. Bala. Software profiling for hot path prediction: less is more. In *Proceedings of the 9th International Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 202 – 211, November 2000.
- [9] K. Ebcioglu and E. R. Altman. Daisy: Dynamic compilation for 100% architectural compatibility. Technical Report RC-20538, T. J. Watson Research Center, May 1996.
- [10] K. Ebcioglu and E. R. Altman. Daisy: dynamic compilation for 100% architectural compatibility. In *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*, pages 26–37, New York, NY, USA, 1997. ACM.
- [11] K. Ebcioglu, E. R. Altman, M. Gschwind, and S. Sathaye. Optimizations and oracle parallelism with dynamic translation. In *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 284–295, Washington, DC, USA, 1999. IEEE Computer Society.
- [12] A. Gal and M. Franz. Incremental dynamic code generation with trace trees. Technical Report 06-16, Donald Bren School of Information and Computer Science, University of California, Irvine, November 2006.
- [13] M. Gschwind, K. Ebcioglu, E. Altman, and S. Sathaye.

- Binary translation and architecture convergence issues for ibm system/390. In *ICS '00: Proceedings of the 14th international conference on Supercomputing*, pages 336–347, New York, NY, USA, 2000. ACM.
- [14] W. Havanki, S. Banerjia, and T. Conte. Tregion scheduling for wide issue processors. In *HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 266–276, New York, NY, USA, 1998. ACM.
- [15] J. D. Hiser, D. Williams, W. Hu, J. W. Davidson, J. Mars, and B. R. Childers. Evaluating indirect branch handling mechanisms in software dynamic translation systems. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 61–73, Washington, DC, USA, 2007. IEEE Computer Society.
- [16] A. Klaiber. *The technology behind CrusoeTM processors*. Tansmeta Corporation, January 2000.
- [17] R. Lougher. JamVM virtual machine, June 2009. jamvm.sourceforge.net.
- [18] B. A. Maher, A. Smith, D. Burger, and K. S. McKinley. Merging head and tail duplication for convergent hyperblock formation. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 65–76, Washington, DC, USA, 2006. IEEE Computer Society.
- [19] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture*, pages 45–54, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.
- [20] W. S. Mong and J. Zhu. Dynamosisim: a trace-based dynamically compiled instruction set simulator. In *ICCAD '04: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 131–136, Washington, DC, USA, 2004. IEEE Computer Society.
- [21] The SPEC Corporation. www.spec.org.
- [22] C. Wang, S. Hu, H. Kim, S. R. Nair, M. Breternitz, Z. Ying, and Y. Wu. Stardbt: An efficient multi-platform dynamic binary translation system. In *Asia-Pacific Computer Systems Architecture Conference*, 2007.
- [23] E. Yardimci. *Exploiting parallelism to improve the performance of sequential binary executables*. PhD thesis, University of Irvine, California, 2006.