Trace Execution Automata in Dynamic Binary Translation

João Paulo Porto and Guido Araujo
LSC - IC - Unicamp
Av. Albert Einstein, 1251 - Cidade Universitária
Campinas, SP, Brasil
{jpporto, guido}@ic.unicamp.br

Edson Borin and Youfeng Wu
PSL - Intel
2200 Mission College Blvd.
Santa Clara, CA, USA
{edson.borin, youfeng.wu}@intel.com

ABSTRACT

Program performance can be dynamically improved by optimizing its frequent execution traces. Once traces are collected, they can be analyzed and optimized based on the dynamic information derived from the program's previous runs. The ability to record traces is thus central to any dynamic binary translation system. Recording traces, as well as loading them for use in different runs, requires code replication in order to represent the trace. This paper presents a novel technique which records execution traces by using an automaton called TEA (Trace Execution Automata). Contrary to other approaches, TEA stores traces implicitly, without the need to replicate execution code. TEA can also be used to *simulate* the trace execution in a separate environment, to store profile information about the generated traces, as well to instrument optimized versions of the traces. In our experiments, we showed that TEA decreases memory needs to represent the traces (nearly 80% savings).

Categories and Subject Descriptors

D.3.4 [**Processors**]: Run-time environments

Keywords

Dynamic Binary Translation, Deterministic Finite Automaton, Trace Recording, Trace Replaying

1. INTRODUCTION

Dynamic Binary Translators (DBTs) rely on information about the dynamic behavior of a program to improve its performance. This is done by detecting and optimizing, code fragments, known as *hot code*, which accounts for the largest share of the program execution time.

To detect hot code, a DBT might use a trace selection technique. Several techniques have been proposed in the literature [1, 5, 7, 13, 17] which address the same issue: how can hot code be easily detected (i.e. with the least possible overhead)? The description of such techniques, as well as their advantages or disadvantages are beyond the scope of this paper, which describes a technique to *record* and *replay* traces.

The *Trace Execution Automata* technique uses a Deterministic Finite Automaton (DFA) to map executing instructions to instructions or basic blocks in previously recorded traces. When operating in *recording* mode, our technique builds a DFA that represents basic blocks (or instructions) from traces. During the *replay* mode, the transition between instructions in the executing program are mapped to transitions in the DFA, which turns into a precise map from the currently executing instructions to the represented basic blocks (or instructions) in the DFA. We found this technique useful in multiple contexts, among them:

- Building traces in one system, e.g. by using a DBT, and collecting statistics and profiling information for them on a second system, e.g. by replaying the traces on a cycle accurate simulator.
- Building and profiling traces without the need for actual trace construction (*e.g* without the need for code replication, code linking and original code patching). This is useful when collecting accurate profiling information before the actual traces code is generated. It is also useful when investigating trace formation techniques because it enables us to focus on the trace formation techniques without concerning about the trace code compilation correctness.
- Storing trace shape and profiling information for reuse in future executions.

This paper is organized as follows. Section 2 presents a motivation for TEA. Section 3 discusses how traces and DFAs are related to each other and shows how to build DFAs out of traces. Section 4 describes our experimental evaluation of TEA. Section 5 lists the previous work on trace recording techniques. Finally, Section 6 concludes the paper and presents the future works.

2. MOTIVATION

Dynamic binary translation usually relies on dynamic profiling information to record and aggressively optimize traces. In this section we show why collecting accurate profiling information before building the actual traces may be challenging.

The code on Figure 1(a) copies one hundred words from the array pointed to by *esi* to the array pointed to by *edi*. Although simple, this optimized code presents a challenge to runtime environments which could eventually optimize it:

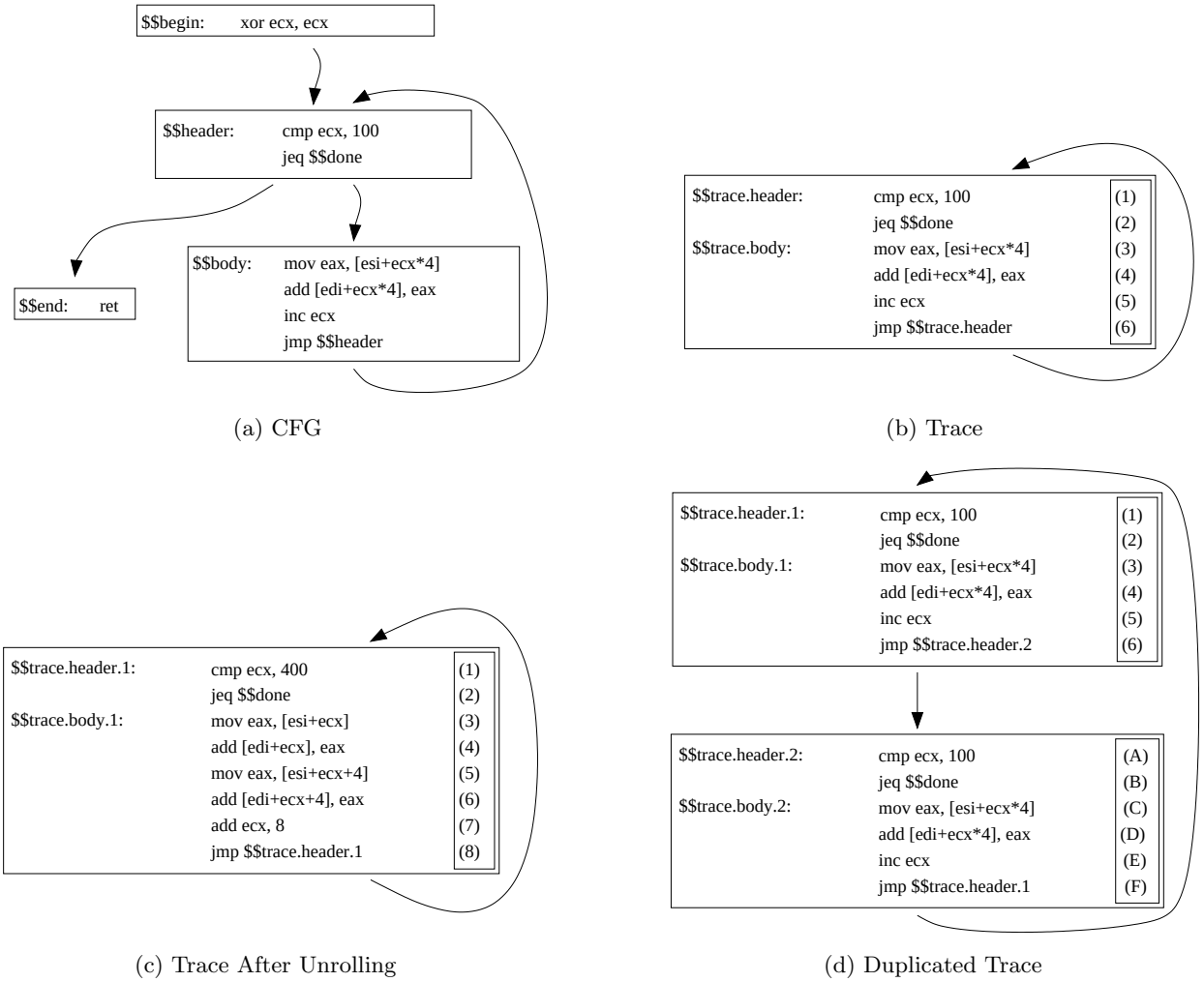

Figure 1: Code Snippet and Resulting Traces

the values in the registers are not known until the application is executed, and might even change across different executions.

Assuming that the code was executed, and the loop it contains was detected as hot code, the trace of Figure 1(b) comes up.

With Algorithm 1 it is possible to create a DFA to simulate that trace's execution. That DFA can now be loaded into a profiling tool (such as our profiling tool described in Section 4) and the profile information for the traces can be gathered.

An obvious question we are yet to answer is why not collecting the profile information as the traces are recorded. The simple, straightforward answer is that it might be easier to implement the trace recording algorithm in an environment where gathering profile data is substantially harder than in another environment. In our experiment, recording the traces was easily done in our DBT environment [21], whereas gathering profile information was easier under Pin [16] as the profile code was ordinary C functions instead of assembly language stubs.

Now, assume that traces are optimized using the profile information collected by replaying the DFA. For example, let's suppose the optimizer unrolled the trace by a factor of two as seen on Figure 1(c). There are now two options to determine the new profiling option.

The first option (the easy one) is to *conservatively* propagate the profiling information for the new instructions. For example, assume that instructions (3) and (4) in Figure 1(b) alias. If this information is conservatively propagated to the unrolled trace, this information is likely to constrain any further optimizations.

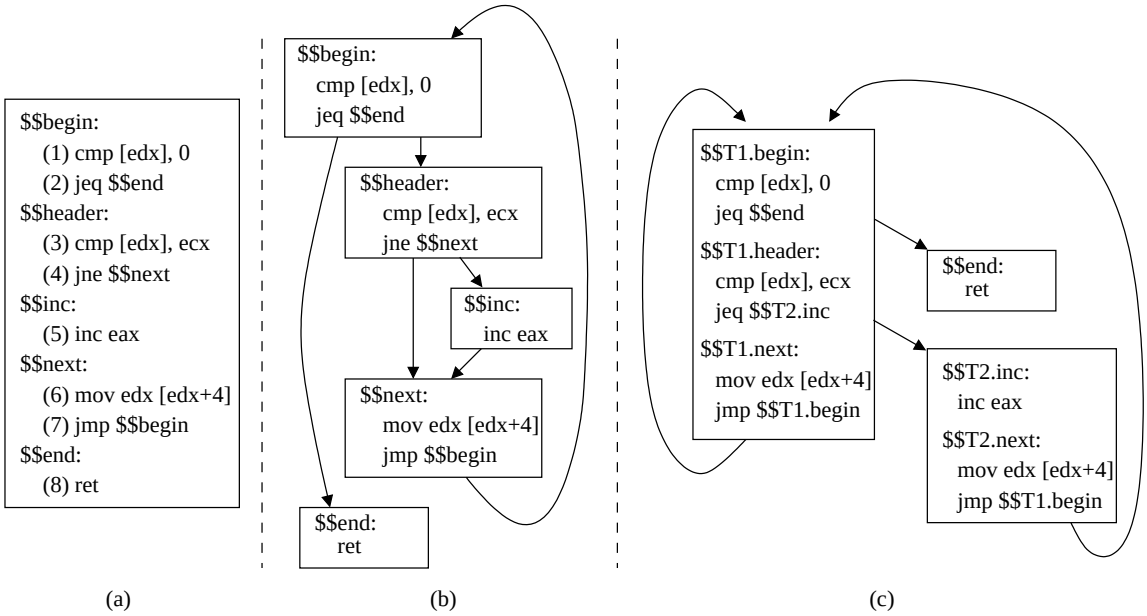

Figure 2: (a) Sample code. (b) CFG for the sample code. (c) MRET traces.

The second, hard option, is to recollect the profiling information. The DFA can not be used to simulate the unrolled loop. Since it does not generate specialized code, the state for the trace would find no corresponding executable code in the executable. However, it is possible to easily work around this: the trace can be **duplicated** instead of **unrolled**. The duplicated trace is shown in Figure 1(d).

The resulting DFA after the trace has been duplicated can be safely loaded alongside the original program for profiling. This new profile data can then be used after unrolling: instructions (C) and (D) in Figure 1(d) are the same as instructions (5) and (6) in Figure 1(c), thus the collected profile information can be used to optimize the unrolled loop. With the new, specialized information the runtime can accurately optimize the code.

The use for the DFA in profiling can be thought as the ability to label duplicate instructions differently for every copy of it in the running program. Code duplication might arise from optimizations such as loop unrolling (as previously illustrated) and function inlining.

3. FROM TRACES TO TEA

In this section we illustrate the relationship between Traces and DFAs and provide an algorithm to build TEA out of traces.

Suppose that a runtime system that builds traces[1] using the MRET strategy [1, 7] is running the compiled code shown, as x86 assembly language, in Figure 2 (a). That piece of code scans a linked list structure pointed to by register *edx* and updates *eax* with the number of times that the value in *ecx* appears on the list.

[1]the word trace will be used from now on as a synonym for hot traces

It might take a few iterations for the runtime system to identify the hot code and invoke the trace recording subsystem. The generated traces heavily depend on the trace selection strategy used as well as on the program's input data. Figure 2 (c) shows two traces (T1 and T2) that could eventually be recorded by using the MRET trace selection strategy. Trace T1 is formed by the basic blocks $$begin, $$header and $$next, while trace T2 is formed by the basic blocks $$inc and $$next.

In our examples, we use the format $$*Ti.block* when referring to a block that belongs to a trace. This format allow us to distinguish blocks that are duplicated (e.g. $$T1.next and $$T2.next) and avoid confusion with the original block name (e.g. $$next).

A trace, or a collection of traces, implicitly defines a DFA. As an example, the DFA for traces on Figure 2 (c) can be seen on Figure 3 (a). Each node in the DFA represents a basic block that is part of a trace. The transitions between nodes represent the control flow in the traces. The label in a transition indicates the address, or the *Program Counter*, that triggers such transition. Notice that the automaton for the trace does not contain the transition from $$T1.begin to $$end. This happens because basic block $$end does not belong to any trace, therefore this transition does not represent control flow inside or between traces.

Suppose that the traces at Figure 2 (c) represents all the hot code for the sample program. To generate a DFA for the whole program all transitions must be accounted by the automaton, including transitions to and from hot code. To model this *whole program* DFA, a special state labeled NTE, which stands for *No Trace being Executed*, is generated. The program is on the NTE state whenever it is not executing any trace. Transitions from NTE to traces are labeled with the traces' start addresses. Transitions from traces to NTE

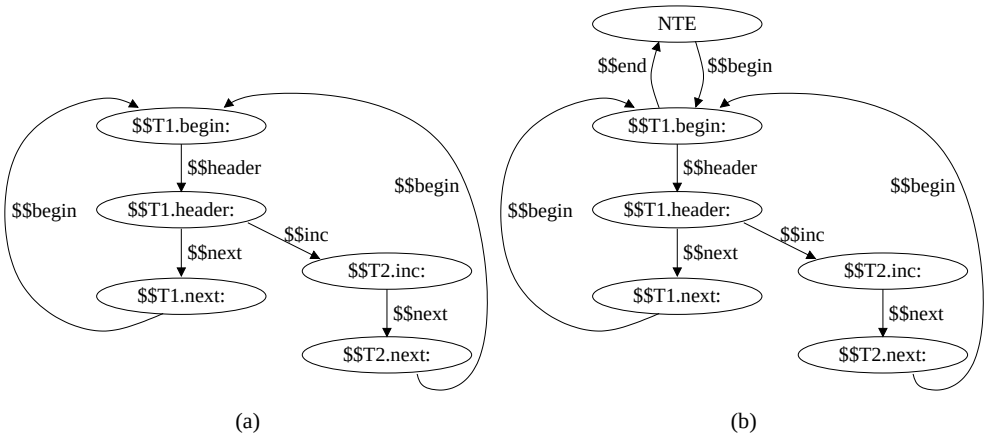

Figure 3: (a) DFA for MRET traces. (b) TEA for whole program.

represent control flow between traces and cold code. We call the *whole program* DFA generated from the execution traces and the NTE state **TEA**. For this sample program, the TEA is represented on Figure 3 (b). The TEA is logically similar to the dynamic control flow graph (DCFG) for the traces seen Figure 2 (c). TEA, however, contains just the *state* information, whereas the DCFG contains code replication. TEA also models the whole program execution with the aid of the NTE state, while the DCFG only represents the hot code.

The generated TEA can be used to replay trace executions without running actual trace code. As an example, we could re-execute the program at Figure 2 (a) on a different system and replay the MRET traces execution by feeding the program counter into the generated TEA. The TEA states provides an accurate mapping from the current program counter to the previously recorded traces. For instance, during the re-execution of the sample program, if the current program counter points to $$next we can precisely tell whether it corresponds to the execution of the original $$next, $$T1.next or $$T2.next by looking at the TEA current state.

The following two sections presents an algorithm that, given a set of traces, builds the corresponding TEA (Section 3.1) and provides some insights on how TEA can be used online to record traces (Section 3.2).

3.1 Building TEA out of Traces

The initial motivation for TEA was the ability to generate traces in one environment and to load and execute them in another. This simple problem becomes hard when the two different environments are extremely different. TEA enabled us to generate the traces in one system, and execute them on another system. Algorithm 1 shows how we converted the generated traces to TEA, but first, some definitions are needed.

DEFINITION 1. *A Basic Block (BB) is a sequence of instructions with a single entry point and a single exit point.*

Usually a BB is terminated by a branch instruction. However, different runtime systems detect basic blocks differently. For instance, on Figure 2 (a) some runtime system might be able to identify block *$$inc* as a basic block, even though it does not end in a branch instruction. Usually, however, DBTs merge blocks *$$inc* and *$$next*. Notice that either way, Definition 1 correctly identifies BBs. Since each BB might be on several different traces and, depending on the recording algorithm, might appear several times on the same trace, there should be a way to uniquely identify each BB on the traces.

DEFINITION 2. *A Trace Basic Block (TBB) is an instance of a BB in a trace. Each occurrence of a BB will generate a unique TBB.*

Given Definition 2, even if BB b occurs several times in the set of traces of a program, it is possible to distinguish between the different *instances* of b. As an example, Figure 2 (c) shows two MRET traces, T1 and T2, with two different instances of BB $$next: $$T1.next and $$T2.next.

DEFINITION 3. *A Trace is a collection of TBBs and the control flow edges between them.*

Definition 3 encompasses many different kinds of traces, from traces made of superblocks to Trace Trees. With the previous definitions, it is possible to explain Algorithm 1 and to prove its correctness.

The first step in Algorithm 1, lines 1 to 2, initializes the TEA with a single state (NTE) and an empty set of transitions. As discussed, the NTE state represents the execution of basic blocks (or instructions) that does not belong to traces. The next step, lines 3 to 5, adds the states to represent the TBBs. Since each TBB is unique, and exactly one state is created for each TBB, the resulting TEA has the following property:

PROPERTY 1. *The resulting TEA is capable of representing the execution of every TBB.*

Algorithm 1: Converting Traces to TEA

Input: Ts: The set of Traces in a Program
Output: TEA: the TEA

```
1  TEA.States ← {NTE}
2  TEA.Transitions ← ∅
3  foreach T ∈ Ts do
4  |  foreach TBB ∈ T do
5  |  |  TEA.States ← TEA.States ⋃ {TBB}
6  foreach T ∈ Ts do
7  |  foreach TBB ∈ T do
8  |  |  foreach Successor S of TBB do
9  |  |  |  if S is a trace block then
10 |  |  |  |  TEA.Transitions ← TEA.Transitions
   |  |  |  |  ⋃
11 |  |  |  |        {TBB → S, Label(S)}
12 |  |  |  else
13 |  |  |  |  TEA.Transitions ← TEA.Transitions
   |  |  |  |  ⋃
14 |  |  |  |        {TBB → NTE, Label(S)}
15 |  foreach TBB ∈ EntryBlocks(T) do
16 |  |  TEA.Transitions ← TEA.Transitions ⋃
17 |  |        {NTE → TBB, Label(TBB)}
18 return TEA
```

The third step, lines 6 to 17, adds the transitions between the states together with the labels that trigger the transitions. First, it adds the transitions that originates at TBBs (lines 7 to 14) by processing the TBBs successor basic blocks. If the successor of the TBB is not a block in a trace, a transition from the TBB to NTE is added, representing a transition from the trace to cold code. Finally, it adds the edges between the state NTE and the TBBs, representing the start of traces execution (lines 15 to 17). Thus the following holds:

PROPERTY 2. *The resulting TEA contains all transitions for every TBB represented.*

Properties 1 and 2 ensure the resulting TEA models the exact behavior of the program's traces, thus proving the algorithm correctness.

3.2 Recording TEA instead of Traces

As previously mentioned, TEA can be used as a online technique for trace recording. It is built by Algorithm 2, which is invoked every time the running program finishes a TBB execution but before the next TBB is executed. Trace recording is expressed as a three-state State Machine. The possible states are "*Initial*", "*Executing*" and "*Creating*", each of which with its own well-defined rule.

State "*Initial*" is executed before the program starts its real execution. It simply sets up an empty TEA (*i.e.* a TEA with only the NTE state) and indicates that the program is in the "*Executing*" state.

In the "*Executing*" state the application is either running cold code or executing a previously created trace. Depending on the trace recording rules (line 7) the state machine switches to the "*Creating*" state.

Algorithm 2: Using TEA to Record Traces

Input: Current: The TBB Previously Executed
Input: Next: The Next TBB to be Executed
Input: State: The Recording Algorithm's Current state

```
1  switch State do
2  |  case Initial
3  |  |  InitializeTEA(TEA)
4  |  |  State ← Executing
5  |  case Executing
6  |  |  ChangeState(TEA, Current, Next)
7  |  |  if TriggerTraceRecording(Current, Next) then
8  |  |  |  StartCreatingTrace(Current, Next)
9  |  |  |  State ← Creating
10 |  case Creating
11 |  |  AddTBBToTrace(Current, Next)
12 |  |  if DoneTraceRecording(Current, Next) then
13 |  |  |  FinishTrace(Current, Next)
14 |  |  |  State ← Executing
```

Trace recording takes place in the "*Creating*" state. Again, depending on the algorithm being used for trace selection, the state machine decides whether or not to end trace recording (line 12).

4. EXPERIMENTAL RESULTS

For this paper, our goals were (1) evaluating how TEA would decrease memory required to represent traces; (2) evaluating how effective TEA is for replaying previously recorded traces on unmodified program executables; and (3) to evaluate TEA's effectiveness as a trace recording tool itself. All the experiments were executed in Ubuntu 9.10 in a virtual machine running under Windows 7 in a Core i7 EE 975 with 12 GB of DDR3 1333 MHz DRAM. Our experimental setup included two different DBT frameworks, pin [16] and StarDBT [21].

Pin is a well-known runtime environment which allows programmers to develop their own profiling tool (called "pintools") composed of instrumentation and analysis routines. Pin offers a rich set of APIs that offers great flexibility. It is indeed a very important tool for binary translation experiments, among other uses. For this paper, we implemented a pintool that loads traces from a input file and uses the traces for program execution. Our tool is also capable of recording traces if they are not available prior to program execution.

StarDBT is a DBT runtime environment which translates IA-32 to IA-32. It is less flexible than Pin, but it offers a greater control over how instrumentation and analysis are done. It was used as a baseline for memory requirements to represent traces. The generated traces were also used by our pintool during the "trace replaying" experiment.

Table 1 shows the data regarding the size needed to represent the traces. We recorded traces using three different techniques (MRET, CTT and TT). Our previous paper on

benchmark	MRET			CTT			TT		
	DBT	TEA	Savings	DBT	TEA	Savings	DBT	TEA	Savings
168.wupwise	329	81	75%	64	14	78%	63	14	78%
171.swim	538	110	79%	998	205	79%	193	38	80%
172.mgrid	671	138	79%	940	198	79%	278	61	78%
173.applu	648	124	81%	1005	187	81%	437	76	82%
177.mesa	583	127	78%	605	126	79%	238	56	76%
178.galgel	1011	238	76%	2083	463	78%	1766	388	78%
179.art	354	90	75%	441	110	75%	322	82	75%
183.equake	442	108	74%	683	157	77%	529	130	75%
187.facerec	674	152	73%	989	211	79%	535	114	79%
188.ammp	551	130	76%	903	197	78%	341	73	78%
189.lucas	113	19	83%	542	103	81%	673	124	81%
191.fma3d	1446	336	77%	1445	294	80%	419	91	78%
200.sixtrack	2162	500	77%	3055	613	80%	1148	225	80%
301.apsi	1346	304	77%	2119	423	80%	695	135	81%
164.gzip	2110	533	75%	51601	11318	78%	598533	143665	76%
175.vpr	1918	457	76%	13893	3093	78%	30687	7298	76%
176.gcc	53203	13147	75%	204203	44728	78%	89358	18917	79%
181.mcf	360	86	76%	855	224	74%	3430	908	74%
186.crafty	1980	493	75%	105018	22224	79%	14829	2998	80%
197.parser	3352	867	74%	25231	5534	78%	17202	3489	80%
252.eon	6217	1007	84%	10218	1677	84%	3732	554	85%
253.perlbmk	17333	4031	86%	78361	16819	79%	48287	9774	80%
254.gap	3183	684	79%	9869	1969	80%	6836	1358	80%
255.vortex	14854	3426	77%	17478	3497	80%	2188	488	78%
256.bzip2	1031	257	75%	59053	13177	78%	1801870	351738	80%
300.twolf	1632	408	75%	9848	2297	77%	7008	1518	78%
GeoMean			77%			79%			79%

Table 1: Size Savings with TEA

traces [17] showed that memory requirements for the three techniques were different from one another. We wanted to evaluate if TEA was sensitive to the technique. The columns labeled "DBT" indicate the memory requirements (in KB) to represent the recorded traces, whereas the columns "TEA" indicate the memory requirements (also in KB) to represent traces using TEA. The "Savings" column indicates the memory usage savings achieved by representing traces with TEA instead of the usual strategy (*i.e.* replicating the code) to be around 80%. TEA achieves this space savings by avoiding code specialization for trace representation.

Table 2 shows the runtime aspects of trace replaying. We again compare our TEA implementation in the Pintool against our "baseline", which are the StarDBT collected traces. The "coverage" columns show how much runtime instructions were executed inside the traces. The "time" column under TEA shows the amount of time needed to replay the traces in our pintool, and under DBT shows the amount of time needed to record the traces in DBT. Since the table displays information about trace *replaying*, it is expected that the coverage for TEA is slightly higher than DBT's coverage since our tool will execute less cold code. This is true for all but one benchmark: 177.mesa. The 0.2% difference in coverage on this particular benchmark occur since Pin and StarDBT use slightly different algorithm to detect individual instructions. Nevertheless, the results are close enough to be considered valid.

Regarding "Time", it is noticeable that TEA presents a somewhat high overhead when compared to DBT's execution. There are at least two reasons for this difference. The first reason is the way Pin inserts the instrumentation code to manipulate the TEA. Usually, pin will insert function calls to the pintool's instrumentation routines, which adds considerable overhead to the program's execution. The other reason is related to TEA's transition function. Every branch instruction is proceeded by a call to a function that eventually searches for the target trace in some sort of data structure. By replicating code to represent the traces, DBT does not need a transition function. The results on this Table (as well as the ones on Table 3) were collected with an optimized transition function. The optimizations are described in Section 4.2.

Table 3 shows the data regarding to our experiment on TEA's ability to record traces. For this experiment, we implemented the MRET [7] trace strategy in our pintool. The columns in the table have the same meaning as they have on Table 2, except "Time" which means "recording time" for both Pin and DBT. Again, the recorded traces present a slightly different coverage and take more time to record. The reasons for the later are the same as the ones for the replaying experiment. The reasons for the former are the difference in how StarDBT and Pin count runtime instructions as well as subtle algorithm implementation differences.

4.1 Implementation Challenges

The most challenging issue faced during the experiments was related to how dynamic basic blocks are identified. StarDBT

Benchmark	TEA		DBT	
	Coverage	Time	Coverage	Time
168.wupwise	100%	2209	100%	151
171.swim	100%	614	100%	100
172.mgrid	100%	802	100%	144
173.applu	100%	725	100%	79
177.mesa	99.8%	1105	100%	87
178.galgel	100%	1412	100%	175
179.art	99.8%	1881	99.5%	110
183.equake	100%	324	100%	38
187.facerec	100%	1189	100%	95
188.ammp	100%	1558	100%	125
189.lucas	90.4%	670	89.3%	86
191.fma3d	94.2%	636	94.1%	98
200.sixtrack	99.1%	1358	99.1%	129
301.apsi	100%	1560	100%	134
164.gzip	99.8%	2913	99.6%	157
175.vpr	100%	1441	99.9%	97
176.gcc	98.1%	2160	97.6%	203
181.mcf	99.9%	635	99.9%	48
186.crafty	95.6%	2058	95.5%	146
197.parser	100%	3482	100%	163
252.eon	91.0%	9417	90.9%	814
253.perlbmk	83.3%	4890	82.9%	253
254.gap	88.3%	2186	87.9%	111
255.vortex	99.4%	3188	99.3%	242
256.bzip2	99.9%	2077	99.9%	117
300.twolf	100%	2977	100%	181
GeoMean	97.5%	1559	97.4%	129

Table 2: TEA Runtime Aspects – Replaying

Benchmark	TEA		DBT	
	Coverage	Time	Coverage	Time
168.wupwise	99.7%	2697	100%	151
171.swim	100%	617	100%	100
172.mgrid	100%	867	100%	144
173.applu	100%	767	100%	79
177.mesa	96.9%	1332	100%	87
178.galgel	100%	1513	100%	175
179.art	100%	1827	99.5%	110
183.equake	100%	308	100%	38
187.facerec	99.3%	1391	100%	95
188.ammp	99.8%	1539	100%	125
189.lucas	100%	667	89.3%	86
191.fma3d	100%	662	94.1%	98
200.sixtrack	100%	1583	99.1%	129
301.apsi	99.2%	1627	100%	134
164.gzip	99.7%	3003	99.6%	157
175.vpr	99.9%	1454	99.9%	97
176.gcc	99.4%	2172	97.6%	203
181.mcf	99.9%	612	99.9%	48
186.crafty	99.7%	2112	95.5%	146
197.parser	100%	3607	100%	163
252.eon	97.5%	15352	90.9%	814
253.perlbmk	99.8%	4407	82.9%	253
254.gap	99.9%	2267	87.9%	111
255.vortex	99.1%	3568	99.3%	242
256.bzip2	99.8%	2168	99.9%	117
300.twolf	100%	2982	100%	181
GeoMean	99.6%	1654	97.4%	129

Table 3: TEA Runtime Aspects – Recording

identifies a TBB as starting at an address which is target of a branching instruction and ending in a branch instruction. Besides this heuristic, Pin also create dynamic basic blocks on some unexpected instructions (*e.g.* x86's cpuid) and instructions with REP prefixes. To address this issues, our pintool inserts the instrumentation code on the taken and fall through edges instead of at the beginning of the TBBs. This guarantees that our pintool will see the same transitions StarDBT saw during trace recording.

Another small issue is related to instruction count. StarDBT counts every instruction to be one instructions, even if it is an instruction with a REP prefix that will iterate some times. Pin, on the other hand, creates a loop for these instructions, and counts each instruction of each iteration as one instruction. For this reason, the number of dynamic instructions seen by StarDBT and Pin are slightly different. This is why Tables 2 and 3 do not show instruction count, but coverage instead.

4.2 Analyzing TEA's Performance

The numbers presented in this paper show that our implementation of TEA poses a heavy overhead for programs. Before collecting these results, we experimented several different implementations for the transition function. This Section describes the changes our pintool underwent to improve its performance.

Table 4 contains six different entries for each benchmark. The first column ("Native") indicates the native performance numbers for the benchmarks. For each benchmark, every entry is normalized with respect to this value, thus all entries in this column being 1.00.

The remaining five entries are all related to program execution under Pin. The column "Without Pintool" indicates the slowdown of running the benchmark under Pin without any pintool loaded. In other words, it indicates Pin's overhead alone, which turned out to be low. Column "Empty" reports the overhead to run the application with TEA with an empty set of traces. For these numbers, no traces were recorded by our Pintool at runtime.

The remaining three columns report the results for loading and replaying traces under three different scenarios. For each benchmark, every experiment use the same set of traces. Column "No Global / Local" indicates that a local cache was used to speed up transitions from one trace to another while no auxiliary data structures were used to speed up trace look up (the traces were kept in a linked list) when the local cache misses. The "Global / No Local" experiment used the global B+ tree to speed up trace look up, while no local caching scheme was employed. The last column, "Global / Local", shows the results when both the global B+ tree and the local cache were used.

The auxiliary structures are very important in the TEA's transition function, which is the responsible for most of TEA's overhead. In fact, the first TEA implementation employed no auxiliary data structures for speeding up trace

Benchmark	Native	Under Pin				
		Without Pintool	Empty	No Global / Local	Global / No Local	Global / Local
168.wupwise	1.00	1.54	43.43	23.57	26.83	19.47
171.swim	1.00	1.04	6.33	4.61	6.15	4.44
172.mgrid	1.00	1.25	5.00	4.12	5.69	3.74
173.applu	1.00	1.09	11.73	6.70	9.90	6.40
177.mesa	1.00	1.25	31.41	29.02	18.61	12.94
178.galgel	1.00	1.06	7.97	5.45	8.33	4.80
179.art	1.00	1.22	30.28	17.05	26.93	18.30
183.equake	1.00	1.15	11.53	6.01	8.94	6.14
187.facerec	1.00	1.27	21.34	18.27	17.47	11.62
188.ammp	1.00	1.05	19.61	9.94	14.79	10.22
189.lucas	1.00	1.12	15.50	7.21	9.84	7.48
191.fma3d	1.00	1.24	10.41	6.52	7.35	5.73
200.sixtrack	1.00	1.00	11.78	6.84	11.10	5.83
301.apsi	1.00	1.11	13.56	11.50	14.44	8.31
164.gzip	1.00	1.34	45.81	22.91	34.46	22.13
175.vpr	1.00	1.18	30.44	16.64	20.72	14.80
176.gcc	1.00	3.93	81.18	278.39	64.43	43.64
181.mcf	1.00	1.04	17.55	9.69	15.65	10.14
186.crafty	1.00	2.60	56.54	51.12	48.96	32.79
197.parser	1.00	2.13	49.02	26.67	39.07	22.10
252.eon	1.00	4.17	62.48	94.77	42.65	30.96
253.perlbmk	1.00	2.97	94.68	60.21	83.72	55.55
254.gap	1.00	2.53	73.82	45.11	57.92	40.04
255.vortex	1.00	2.30	70.89	223.68	44.22	30.63
256.bzip2	1.00	1.51	37.17	20.24	27.76	18.93
300.twolf	1.00	1.15	30.34	16.98	28.10	17.49
GeoMean	1.00	1.50	25.27	18.52	20.33	13.53

Table 4: TEA Overhead for Various Configurations

look up. The numbers for this particular experiment (which would be the "No Global / No Local" column in Table 4) were not collected since the slowdown was over 2 orders of magnitude from the native execution.

Our first attempt to speed up the transition function was the global B+ tree. The results were interesting, but the overhead was still very high. Later, we implemented the local cache to avoid going to the global trace container every time the system needed to search for a trace. Again, the results improved over the previous data. This configuration ("Local / Global") was used to collect all the data for the Recording / Replaying experiment.

We also investigated whether or not the global B+ tree was important to the overall performance. The experimental data shows that, while the local cache is more important than the global B+ tree, the B+ tree is important as well. A comparison between columns "No Global / Local" and "Global / Local" clearly indicates a performance improvement when using the more optimized global container. Particularly, GCC and Vortex experience a severe slowdowns without the global indexing structure.

The data on the "Empty" column report a counter intuitive result. Having no traces to simulate should be faster than having several traces. However, the numbers do make sense, as the transition function is optimized for the common case (*i.e.* executing hot code). TEA performs more work to switch to cold code than it does for hot code transitions. This run had the global B+ tree and did not have any local caches (local caches are pointless outside of traces in our implementation anyway).

5. PREVIOUS AND RELATED WORK

Traces are closely related to dynamic binary translation and dynamic compilation techniques. Suganuma *et* al. [20] presented a complex JIT compiler for a production level Java Virtual Machine. Unlike previous approaches, that used method boundaries for JITing, they implemented a multi-level compilation strategy and use dynamic compiler to dynamically form "regions", which are their runtime system's compilation unit. Zaleski *et* al. [23] presented an extensible JIT compiler which uses traces as compilation units.

Several trace recording strategies exist on the literature. MFET[2] [5] instruments edges in the dynamic program execution to detect frequently executed paths. MRET[3] [1, 7] instruments back edges only, thus posing less runtime overhead than MFET. TT[4] [13] record traces which always end with a branch to an "anchor", generally a loop header. CTT[5] [17] tries to address the code duplication experienced by TTs by allowing branch targets within a path to be any

[2]Most Frequently Executed Tail
[3]Most Recently Executed Tail
[4]Trace Trees
[5]Compact Trace Trees

loop header in that path.

Another use for traces in JIT compilers is described by Gal *et* al. [12]. They use Trace Trees [13] as compilation units for the SpiderMonkey JavaScript Virtual Machines. Besides all the complication in a JVM JIT compiler, the authors face more challenges since JavaScript is dynamically typed.

Besides those well-known uses, recently Wimmer *et* al. [22] used traces to perform phase detection. A program phase is identified when the created traces are *stable* (*i.e.*, there is a low trace exit ratio). Whenever program execution start to take side exits more often, the program is said to be in an *unstable* (*i.e.* between phases).

Several well-known optimization systems have employed traces to capture program's code locality. Examples of these environments are Dynamo [2], FX!32 [4] and the IA-32 Execution Layer [3].

All the previously mentioned systems work with user mode code. More complicated DBT systems can translate system level code. For instance, DAISY [8, 9, 10, 14] is a compatibility layer which translates PowerPC code to a underlying VLIW systems. The Transmeta CMS [6] is the compatibility layer on the top of the Crusoe [15] micro processor. They both utilize some sort of trace recording to detect hot code. Both system could have applied TEA as a tool for dynamic trace recording.

On the hardware side, traces have been used for high-bandwidth instruction fetch [18]. The Pentium IV processor [11] implements a trace cache. High-bandwidth instruction fetch is achieved since *logically* contiguous instructions in the instructions stream are placed adjacent to one another in the Trace Cache. This high-bandwidth cache was needed due to the high clock frequencies that processor achieved [19]. TEA is not related to trace caches since it does not require instructions to be contiguous on the instruction stream.

6. CONCLUSIONS AND FUTURE WORK

This paper presents TEA, a technique that uses Deterministic Finite Automata (DFA) to map executing instructions to instructions or basic blocks in previously recorded traces. We list multiple contexts in which TEA is useful and we discuss the implementation challenges and solutions when implementing TEA on StarDBT and Pin frameworks.

Our experimental results show that the resulting TEA's transition lookup operation plays a fundamental role on TEA's performance. For this paper, we implemented the lookup operation with the help of a auxiliary look up data structures, which is searched whenever there is a transition from cold code to hot cold, or when there is a transition from one trace to another. In the future, we will investigate other techniques to optimize the transition lookup operation and amortize TEA's cost.

7. REFERENCES

[1] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *SIGPLAN Not.*, 35(5):1–12, 2000.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, 2000.

[3] 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.

[4] A. Chernoff and R. Hookway. Digital fx!32 running 32-bit x86 applications on alpha nt. 1997.

[5] C. Cifuentes and M. V. Emmerik. Uqbt: Adaptable binary translation at low cost. *IEEE Computer*, pages 60 – 66, March 2000.

[6] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The transmeta code morphing^{TM}software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 15–24, Washington, DC, USA, 2003. IEEE Computer Society.

[7] E. Duesterwald and V. Bala. Software profiling for hot path prediction: less is more. In *Proceedings of the 9^{th} International Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 202 – 211, November 2000.

[8] K. Ebcioğlu and E. R. Altman. Daisy: Dynamic compilation for 100% architectural compatibility. Technical Report RC-20538, T. J. Watson Research Center, May 1996.

[9] K. Ebcioğlu 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.

[10] K. Ebcioğlu, 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.

[11] D. H. Friendly, S. J. Patel, and Y. N. Patt. Putting the fill unit to work: dynamic optimizations for trace cache microprocessors. In *MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 173–181, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[12] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 465–478, New York, NY, USA, 2009. ACM.

[13] 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.

[14] M. Gschwind, K. Ebcioğlu, 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.

[15] A. Klaiber. *The technology behind Crusoe^{TM} processors*. Tansmeta Corporation, January 2000.

[16] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[17] J. P. Porto, G. Araujo, Y. Wu, E. Borin, and C. Wang. Compact trace trees in dynamic binary translators. In *2nd workshop on architectural and micro-architectural support for binary translation (AMAS-BT'09)*, June 2009.

[18] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 24–35, Washington, DC, USA, 1996. IEEE Computer Society.

[19] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. *SIGARCH Comput. Archit. News*, 30(2):25–34, 2002.

[20] T. Suganuma, T. Yasue, and T. Nakatani. A region-based compilation technique for dynamic compilers. *ACM Trans. Program. Lang. Syst.*, 28(1):134–174, 2006.

[21] 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.

[22] C. Wimmer, M. S. Cintra, M. Bebenita, M. Chang, A. Gal, and M. Franz. Phase detection using trace compilation. In *PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 172–181, New York, NY, USA, 2009. ACM.

[23] M. Zaleski, A. D. Brown, and K. Stoodley. Yeti: a gradually extensible trace interpreter. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pages 83–93, New York, NY, USA, 2007. ACM.