Efficient Generation of Compact Execution Traces for Multicore Architectural Simulations

AYMAN HROUB, M. E. S. ELRABAA, M. F. MUDAWAR, and A. KHAYYAT,
King Fahd University of Petroleum and Minerals

Requiring no functional simulation, trace-driven simulation has the potential of achieving faster simulation speeds than execution-driven simulation of multicore architectures. An efficient, on-the-fly, high-fidelity trace generation method for multithreaded applications is reported. The generated trace is encoded in an instruction-like binary format that can be directly “interpreted” by a timing simulator to simulate a general load/store or x8-like architecture. A complete tool suite that has been developed and used for evaluation of the proposed method showed that it produces smaller traces over existing trace compression methods while retaining good fidelity including all threading- and synchronization-related events.

CCS Concepts: • **Computing methodologies** → **Simulation tools**; • **Computer systems organization** → **Multicore architectures**;

Additional Key Words and Phrases: Execution traces, trace compression, trace-based simulations, multicore simulations

ACM Reference format:

Ayman Hroub, M. E. S. Elrabaa, M. F. Mudawar, and A. Khayyat. 2017. Efficient Generation of Compact Execution Traces for Multicore Architectural Simulations. *ACM Trans. Archit. Code Optim.* 14, 3, Article 27 (August 2017), 25 pages.
<https://doi.org/10.1145/3106342>

1 INTRODUCTION

Exploring different many-core architectures through simulations represents a huge challenge for both architects and application developers. This is due to the complexity of these new architectures with their many cores, interconnection network, and complex cache hierarchies. Many approaches have been proposed to simulate such complex architectures with adequate speed and fidelity. The most widely used approach is to parallelize software simulations of multicores using multiple simulation threads that run on multicore machines (Argollo et al. 2009; Jun et al. 2014; Miller et al. 2010; Pengju et al. 2012). Another approach for accelerating many-core simulations is to use Field-Programmable Gate Arrays (FPGAs) (Khan et al. 2012; Tan et al. 2010). FPGAs offer massive

This work is supported by the Deanship of Scientific Research at King Fahd University of Petroleum and Minerals, Saudi Arabia, under the NSTIP project #12-INF3015-04.

Authors’ addresses: A. Hroub, M. E. S. Elrabaa, M. F. Mudawar, and A. Khayyat, Computer Engineering Department, King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia; emails: {aymanh, elrabaa, mudawar, akhayyat}@kfupm.edu.sa.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2017 ACM 1544-3566/2017/08-ART27 \$15.00

<https://doi.org/10.1145/3106342>

fine- and coarse-grained configurable logic fabric that can emulate the simulated architecture or a simplified model of it. Other approaches utilize abstract models of the simulated architecture to accelerate the simulations or combinations of several approaches (Carlson et al. 2011).

The aforementioned techniques utilize execution-driven simulations, requiring a detailed microarchitecture modeling of the target machine. This could be very taxing in terms of area and resources for hardware-based simulators (that use FPGAs). Trace-driven simulators, on the other hand, do not require detailed microarchitecture modeling since no actual execution is required. A trace is generated once using actual execution on a real machine or functional simulation and is then used to analyze/evaluate a target machine's performance, memory subsystem's performance, branch prediction and prefetching efficiency, and so forth. Trace-driven simulation of computer systems has been widely used among computer architects and application developers (Prete et al. 1995). This is due to its convenience and ease of implementation. It can reveal considerable useful information, such as an application's average clocks per instruction (CPI), cache performance, locality of references, efficiency of branch prediction, and prefetching. A typical trace comprises the executed instructions along with their corresponding memory references. A trace-driven simulator can be a complete simulator for the whole computer system or specific for a certain component, such as a branch predictor or instruction cache. Trace fidelity refers to how many of the original execution events can be reconstructed from the trace.

In the many-core era, researchers paid more attention to execution-driven simulation than trace-driven because traditionally, trace-driven simulators do not capture time-dependent thread execution and synchronization behaviors. However, researchers and architects continued to use trace-driven simulation to simulate multithreaded applications on multicore machines (Butko et al. 2015; Nilakantan et al. 2015; Rico et al. 2011).

Another major challenge of trace-driven simulation is the large size of trace files. Although disk storage is currently inexpensive, the disk access time is still high. Moreover, the situation is not improved when FPGAs are used for trace-driven simulation due to their limited storage resources. Although existing trace compression techniques succeeded in achieving excellent compression rates, these techniques still suffer from two drawbacks: (1) most of these techniques take the full original trace as input (they do not compress on the fly), and (2) many require a decompression stage before the original large trace can be reproduced or used as input (e.g., Burtscher et al. (2005) and Kanev and Cohn (2011)).

An alternative to trace compression is to choose a representative subset of an application to evaluate the performance of a computer system (Yi et al. 2006). Simulation with the selected subset might produce an average CPI similar to the original application; however, many execution details of the original application are lost. Hence, there is a need to generate compact execution traces in a format that retains fidelity as much as possible and can be directly "interpreted" by a software or FPGA-based architectural simulator (i.e., no or very little decompression effort). The generated trace should be as architecture agnostic as possible; though it is generated from execution on one architecture, it can be used to simulate other architectures.

This article presents a novel methodology for efficiently generating compact execution traces of multithreaded applications on the fly (i.e., during the application execution or functional simulation). With a simple structure that can be efficiently stored and processed, the compressed trace can be used for both software- and FPGA-based simulators. A special compact execution trace (CET) format has been developed. It encodes most of the application's low-level execution events, including thread creation and synchronization events, with minimum size and can be processed directly without decompression. Hence, a trace-based timing simulator can reconstruct most of the user mode execution events in the correct order. In addition, a complete tool suite that generates the CET trace has been implemented and used to evaluate the proposed methodology.

Next, Section 2 presents a survey of related work followed by the details of the basic strategy and methods developed to generate CET code in Section 3. This includes a description of the chain of phases the input application undergoes to generate the compact trace. Experimental results using the developed tool suite are presented in Section 4. Finally, conclusions and future work are presented in Section 5.

2 RELATED WORK

An address trace compression technique based on loop detection was proposed in [Elnozahy \(1999\)](#). Control-flow analysis (scanning the trace and finding repeated patterns) is used to detect loops that have constant or varying-by-constant addresses. Complex situations in which loops have function calls and/or complex structures were not handled. In [Milenkovic and Milenkovic \(2003\)](#), a stream-based compression (SBC) was proposed. Stream blocks (instructions between branching) are replaced by assigned indices, exploring locality. However, when the number of stream blocks grows, the compression ratio decreases. Later on, the authors further improved their SBC method and utilized variable field-length encoding/decoding ([Milenkovic and Milenkovic 2007](#)). SBC then achieved better compression ratios than PDATS ([Johnson et al. 2001](#)), LBTC ([Luo and John 2004](#)), and TCgen ([Burtscher et al. 2005](#)). To achieve more compression, the original trace's, a Dinero trace ([Edler and Hill 1998](#)), instruction and data addresses were separated into two files and removed the 2-bit headers that indicated the type of address. This boosted the compression ratio since the headers were noncompressible, but significant details were lost, which limits the use of such traces in simulations. In [Chen et al. \(2013\)](#), a lossless trace compression technique that exploits spatial and temporal locality was proposed. The technique is limited to instructions and their addresses only; that is, data addresses are not covered. Instruction addresses were classified into two categories: (1) sequential addresses with a constant difference between any two consecutive addresses and (2) nonsequential addresses with variable address strides. The input trace consists of pairs of numbers: the instruction address and the instruction itself. The output consists of three components: the static program, the sequent address file, and nonsequential addresses. The second file consists of a very-long-bit vector, where each bit corresponds to a trace element to indicate whether the address is sequential or not. The authors have compared their compression rate to those of PDATS, PDI ([Johnson et al. 2001](#)), LBTC ([Luo and John 2004](#)), and SBC ([Milenkovic and Milenkovic 2003](#)). Their technique outperformed all of these techniques.

[Budanur et al. \(2012\)](#) proposed a memory trace compression technique for SPMDs (single program multiple data). It is based on PRSD (power regular section descriptor) abstractions ([Marathe et al. 2003](#); [Noeth et al. 2009](#)), but it is finer grained. They called it EPRSD (extended PRSD). A PIN-based instrumentation tool (*memtrace*) takes an SPMD application as input and generates its memory trace on the fly. The generated trace is compressed using EPRSD. The *memtrace* tool runs as a set of MPI processes. Each process instruments an SPMD program and outputs the trace into a pipe. The trace compressor consumes the trace from the pipe. The compressor performs intrathread compression on the fly utilizing the repetitive patterns. After instrumentation terminates, it performs interthread compression by factoring out the common parts among threads and finally performs interprocess merging among all processes of the SMPD application. It reduced the trace size by half for the AMG benchmark. Unlike our technique, this technique requires a decompression phase. [Janapsatya et al. \(2007\)](#) proposed a trace compression technique for instructions' addresses alongside an instruction cache analysis method. Their main objective was not to maximize the compression ratio but to accelerate trace processing. This technique is limited to instructions' addresses only. Their technique achieved a simulation speed up to 9.67 over existing techniques, but the compression ratio was 2 to 10 times worse than Gzip. In [Burtscher et al. \(2005\)](#), four VPC (value prediction-based compression) algorithms were introduced, namely, VPC1, VPC2,

VPC3, and VPC4. The input trace consists of pairs of numbers; the first is a 32-bit PC, and the second is a 64-bit extended data (ED). VPC algorithms use predictors to predict the next value based on the previously observed values. If the next value is predicted correctly, the index of the predictor that predicts it is retained. The unpredicted values are outputted to a different stream. If more than one predictor predicts a certain value, there are heuristics to select the best one. For example, VPC1 uses Huffman encoding. If more than one predictor is correct, then the shortest Huffman code is selected. Because the number of predictors is small, the number of bits to encode the predictor's index is smaller than the corresponding trace element. Therefore, the trace is compressed. The same algorithm is applied in the reverse manner to decompress the compressed trace. The TCgen (Burtscher et al. 2005) is a VPC tool that autogenerates a value-prediction-based trace compressor based on user specifications. In Asanovic et al. (2005), a technique for directory and cache state reconstruction (similar to warming up) to accelerate sampled multiprocessor simulation is proposed. A software structure called MTR (Memory Timestamp Record) that can be updated during fast-forwarding (functional simulator that updates the architectural state in between sampling points) is used. For each memory block (cache block), there is an MTR record that registers the ID of the last processor that modified this block, the timestamp of the last write operation, and an array of timestamps of the read operations on the block (each timestamp per processor). During fast-forwarding, a read/write operation will update the MTR record. The directory and cache state reconstruction occurs right before each sampling point. This is done in two steps: (1) determining the subset of blocks that are still cached and (2) checking cross-processor interactions to determine which of these blocks should be valid or dirty according to the cache coherence protocol. This technique, however, works for sampled execution-driven simulations but not for trace-driven simulations. In Barr and Asanovic (2006), a technique to compress branch trace information to be used in snapshot-based microarchitecture simulation is introduced. The compressed trace can be used to warm any arbitrary branch predictor's state before timing simulation of the snapshot. However, this technique is specific for branch information. A trace compression algorithm based on simple loop detection using data addresses was proposed in Ketterlin and Clauss (2008). A data address trace is scanned to detect loop nests using the linear progressions of the addresses. The output is a sequence of loop nests. This algorithm can handle simple loops only and is limited to data addresses. In Zhang and Gupta (2005), a unified trace representation dubbed whole execution traces (WETs) is proposed. WET was constructed by labeling a static program representation with profile information, which is then compressed in a two-step methodology that allows efficient traversal of the generated trace to extract information corresponding to individual profile types. The achieved compression rate was between 16 and 83. In Nilakantan et al. (2015), the Valgrind dynamic binary instrumentation tool is used to generate traces for multithreaded applications that abstract the original instructions into three types of "events": computation (work performed local to a thread), communication (read/write dependencies between threads), and synchronization (embedded pthread calls for each thread). A separate trace is generated for each thread. The computation events contain counts of integer/floating-point operations, memory write/reads, and a list of their unique virtual memory addresses. No control flow events are captured.

Butko et al. (2015) proposed using trace-driven simulations for multicores. Traces (one file/core) are collected from GEM5 multicore system simulations by capturing and storing all requests (core side) and responses (memory side) at the interface of each core's private cache memory. Again, no control flow is captured in the traces, and only trace information related to cache miss events is kept as only those result in external memory traffic (their objective of trace collection).

Rico et al. (2011) also proposed a trace-driven methodology for multicore simulations. They capture full raw traces (no compression or compaction) of sequential sections of tasks or threads with calls to parallelization events (task/thread) spawning events. Other trace generation techniques

such as Bodik et al. (2006), Bodik et al. (2003), and PinPlay (Patil et al. 2010) are concerned with deterministic replay of the program by recording a fixed execution path for the nondeterministic events (e.g., thread interleaving and memory operation order). This deterministic replay is useful for software debugging and computer architecture simulation. However, since replay implies real execution of the program, these techniques do not work for trace-driven simulators because real execution requires functional units that are missing in such simulators.

3 CET GENERATION

The proposed trace generation method in this work translates a multithreaded input application's executable into another binary format called CET code. The latter encodes the original application static code and the data required for timing simulation in a compressed format. The data that cannot be compressed (i.e., embedded into the CET code) is kept aside and is called CET data. So, each thread of the application is translated into five files, namely, the CET code, branch results, jump displacements, loop counters (in the case of inner loops whose counters do not follow a certain pattern), and data addresses (for nonuniform data referencing). Interthread compression was not pursued in the current work to maintain the simplicity of the compressed trace (and hence eliminate decompression). The generated compact trace is only intended for simulation and not for debugging. It can be used by architectural/timing simulators to perform deterministic replay of the same trace on different architectures. It captures the multithreading synchronization events with CET-defined primitives to create, pause, resume, and terminate threads. These primitives are used to encode synchronization barriers, access to critical sections (locks/unlocks), and atomic read-modify-write operations in the generated CET trace. The resulting CET code size is still less than the application's executable size. The CET data file size varies depending on the application. CET code and data are generated once for a specific input program and dataset, but can be used to simulate many architectural configurations (general LOAD/STORE and x86-like core architectures). The CET code needs to be regenerated only when the number of threads changes. No timing information is recorded into the CET code as these will differ with different architectural configurations. For exact replay of multithreaded traces, PinPlay (Patil et al. 2010) can be used.

3.1 Basic Strategy

The basic strategy in the proposed trace generation and compaction methodology is to remove all possible redundancies, in both instructions and data memory references, from the execution trace while preserving a record of as many of the execution events as possible (i.e., fidelity) and maintaining simplicity for the generated compact trace. A static executable code (CET code) is generated along with the minimum execution data (CET data) from which most of the execution events can be reconstructed by a simulator. Several strategies have been employed to achieve these goals:

- (1) **Loop detection:** This is the main strategy to produce static CET code that resembles the original application's executable code. The developed algorithm does that very well for all types of loops (simple/nested with constant/regular/variable counters).
- (2) **Data/instruction addresses compression:** Loop detection is just the first step; if all data/instruction references in the loop body are kept as is (for fidelity), then there will be no compression. Therefore, our strategy was to detect various types of data/instruction references and keep minimum information about them required to recalculate them during simulations. Hence, constant references or references with constant strides are encoded in the CET code itself. Furthermore, all the irregular references stored in the CET code and/or data are relative to previous references (i.e., only the difference from the previous reference is stored). CET data is generated as a FIFO, with the same order as the CET

instructions that utilize them. Hence, the original trace can be readily reconstructed and simulated from both the CET code and CET data using a thread's initial instruction address.

- (3) **Use minimum field sizes:** All fields in the CET code and data (except for CET op-codes) are under the direct control of the user and can be set to the minimum required.

The generated CET code and data have the following features:

- (1) The original program's execution order (control flow) is preserved without keeping any instructions' addresses except the initial thread address. The CET framework supports SIMD Load/Store instructions. They appear in the instrumented trace loads/stores with contiguous data addresses. This allows using the CET trace in simulating architectures that do not support SIMD. The simulator can treat load/store instructions as separate instructions with contiguous data addresses or as SIMD instructions. SIMD ALU instructions are currently treated as regular ALU instructions. It should also be noted that memory fencing instructions (that enforce a certain memory load/store order) will not affect the execution trace, which is always generated in order. Memory fencing instructions can be inserted in the generated trace when it is used in an architectural simulator that supports out-of-order execution.
- (2) The CET code and data format simplifies their generation and usage in simulations, specifically:
  - All the fields in the CET instructions have fixed lengths. This is crucial for FPGA-based simulators where the CET trace would be stored in an on-chip or on-board memory. Having fixed-width fields simplifies the code storage, fetching, and decoding.
  - CET instructions that require no data or utilize CET data consist of the opcode field only.
  - For other instructions, the required data is embedded in the CET instruction itself (e.g., the address of scalar LOAD/STORE instruction, the base address, and the stride value in the contiguous LOAD-C/STORE instructions, the LOOP counter, etc.).
- (3) The CET data files (for each thread) are:
  - a. **CET Addresses:** These are the noncontiguous data load/store addresses. Only the difference from the previous address is encoded in the CET data, not the complete address, reducing their total size by at least 50%. The user specifies the size of this field (default is 16 bits).
  - b. **Branch Results:** These are the results of conditional branch instructions (taken or not taken), when they are executed multiple times in a loop body (but not to implement the looping itself). The size of this field is 1 bit.
  - c. **Jump-M:** Dynamic target addresses of unconditional jump, call, and return instructions, when the jump is to multiple locations. Only the displacement (in number of CET instructions) between the current instruction and the target instruction is stored. The user specifies the size of this field (default is 16 bits).
  - d. **Loop-R:** Irregular inner-loop counters (number of iterations). This is for inner loops that have a different number of iterations per outer loop iteration, and these counters do not follow a certain pattern. Again, the user specifies the size of this field (default is 32 bits).

A special tool has been developed to verify the effectiveness of the proposed CET code generation methodology. It can be integrated with the trace generator (i.e., the functional simulator) or the instrumentation tool. This facilitates the start of the compact trace generation on the fly (i.e., while the program is being executed or emulated), making our method extremely efficient in terms of time and memory requirements. Instructions and function calls in the original execution

Fig. 1. CET tool workflow.

trace are classified into one of **18** unique categories that belong to **six** different classes. These 18 categories are agnostic to any LOAD/STORE or x86-like general-purpose core architecture. They were chosen to be the minimum set that can be used by a trace-driven simulator to reconstruct execution events. Figure 1 shows the three phases of CET generation. The input is an executable file of the multithreaded program alongside its input data that goes through a three-tool chain: the profiler, the code generator and the emulator and CET data generator. Using PIN instrumentation, these phases are repeated for each thread in the input program. The final output of the tool comprises the CET code and data for each thread separately. Producing separate CET codes and data for threads allows simulators to process these threads in parallel. Moreover, the CET tool generates a log file of useful information for the user. It also generates the starting address of each thread, which represents the point of code at which the thread has to be created. Intel’s Pin framework (Intel 2012) has been used for instrumentation in the current version of CET tool which restrict it to work in the user-mode only Other ISAs can be supported using instrumentation tools such as Valgrind (2000). A full-system functional simulator can also be used to generate system-level code traces. It should be noted that such events as thread-interleaving won’t affect capturing a thread instructions since each thread’s trace is kept separate. So e.g. if a loop trace in a certain thread is interrupted due to thread interleaving, the trace will be continued when the thread resumes and the generated trace would not show the interleaving. Each category is assigned a unique CET code and has special arguments. Table 1 summarizes the different instruction classes, categories, their CET code format, and their corresponding CET data (if any). In addition to the CET codes’ formats shown in Table 1, the CET code contains the register numbers of the corresponding original instructions, allowing the simulator to capture hazards in the CET code. For the **System calls** class (all other system calls not related to synchronization), the unique system call identifier/number is encoded in the CET code using 10 bits. This is sufficient for all existing operating systems where the number of system calls does not exceed 500 (“Searchable Linux Syscall Table for x86 and x86_64”). The total number of CET instructions is **26**, leaving **six** unused codes for new CET instructions if needed.

3.2 CET Profiler

In this phase, the application’s profile is generated using functional simulation or native execution. Using instrumentation, information about executed instructions is collected: memory references accessed by loads/stores, branch results in the case of conditional branch, and so forth. The generated application’s profile is an intermediate image of the application where each executed instruction is an object (organized as one object array per thread). Each object contains an opcode, a list of memory references, a list of loop counters, a list of conditional branch results (taken/not

Table 1. CET Code and Data Format for Different Instruction and Function Call Classes and Categories

CET Code	CET Code Format	CET Data*	Description
Unconditional Branch Instructions			
JUMP	5 bits Op_code 16 bits Displacement	None	Jump/call/return instructions that always jump to the same target address
JUMP-M	5 bits Op_code	Jump's displacement	Jump/call/return instructions that jump to different targets
Conditional Branch Instructions[†]			
BRANCH	5 bits Op_code 16 bits Displacement 1 bit BR	Branch result (taken/not taken)	Normal conditional branch instruction. The BR bit records whether the branch was taken
LOOP	5 bits Op_code 16 bits Displacement 20 bits Counter	None	Loop instruction that always has the same counter (Number of iterations)
LOOP-C	5 bits Op_code 16 bits Displacement 20 bits Counter 3 bits INC		Inner loop instruction whose counter differs by constant (INC) for each outer loop iteration
LOOP-R	5 bits Op_code	Loop's counters	Inner loop instruction whose counter differs by a random value for each outer loop iteration
Memory Instructions			
LOAD/STORE	5 bits Op_code 32 bits Address	None	Load/store instruction that accesses the same memory location every time it is encountered
LOAD/STORE-C	5 bits Op_code 32 bits Address 3 bits INC	None	Load/store instruction that accesses a contiguous block of data in memory (e.g., vector). INC is the size of the data element
LOAD/STORE-NC	5 bits Op_code	Data addresses	Load/store instruction that accesses a noncontiguous (scattered) block of data in memory (e.g., dynamic data structure)
Synchronization Function Calls			
START		None	Start a new thread
PAUSE	5 bits Op_code 10 bits Thread Id	None	Pause a thread
WAKE	5 bits Op_code 10 bits Thread Id	None	Wake a sleeping thread
TERMINATE	5 bits Op_code 10 bits Thread Id	None	Terminate a thread

(Continued)

Table 1. Continued

CET Code	CET Code Format	CET Data*	Description
ALU Instructions			
INT-ALU	5 bits Op_code	None	Integer ALU instruction
FP-ALU	5 bits Op_code	None	Floating-point ALU instruction
MULTIPLY	5 bits Op_code	None	
DIVIDE	5 bits Op_code	None	
SYS_CALL	5 bits 10 bits Op_code Sys Call Number	None	System calls other than thread-related calls.

*Stored in three files: (1) jump displacements, (2) branch results (taken/not taken), and (3) loop counters and data addresses.

†Loops are categorized and encoded as conditional branches.

For each application thread, Repeat the following steps after each instruction *I* is executed:

1. If *I* does not exist in the profiled image, create a new object of *I* with the appropriate opcode and add it to the image.
2. If *I* is load/store, add the data address to the list of addresses of *I*.
3. If *I* is conditional branch, add the branch result (T/NT) to the branch results' list.
4. If *I* is an unconditional jump, call, or return, add its target address to the addresses' list.
5. If *I* is an explicit loop, increment its counter.

Fig. 2. CET profiling algorithm.

taken), and so forth. For each instruction type, the corresponding list is filled during profiling. The input application is profiled dynamically by instrumenting each instruction; when an instruction is encountered for the first time, a new object for this instruction is created and mapped to a unique location in the profiled image. If the same instruction address is encountered again later, its corresponding object is updated if required. Figure 2 shows the CET profiling algorithm.

3.3 Loop Recognition Algorithm

x86 architecture has multiple explicit loop instructions, namely, LOOP, LOOPE, LOOPNE, LOOPZ, and LOOPNZ. These instructions are directly detected by the CET profiler and encoded into CET loops. However, compilers often use the conditional branch instructions to encode loops. Therefore, there is a need to distinguish between the conditional branch instructions that implement loops and other conditional branches. Loops represent the main venue for an execution trace compaction. Moreover, detecting the x86 conditional branches that implement loops and translating them into CET loops will minimize the size of CET data significantly. For example, if all x86 conditional branches (CBs) are left as they are, then a loop of 1 million iterations will require storage for 1 million bits to store its branch's results (taken or not taken). However, with loop detection, this branch instruction is translated into a one-CET-loop instruction whose number of iterations is embedded into its body. Conditional branches implementing loops are distinguished from other

Fig. 3. Branch results' chain of a loop candidate x86 conditional branch instruction.

1. Let L be a linked list of loop entries.
2. The loop entry E is a structure consisting of the loop instruction address and counter.
3. When a loop or loop candidate I is encountered, do the following:
  - a. Let E be the loop entry of this I .
  - b. If E does not exist on L , push it on the front of L .
  - c. Else, if E is the front element of L and its counter is not zero, decrement the counter.
  - d. Else, if E is the top element of L and its counter is zero, pop L off.
  - e. Else, if E exists in L but not the front element, I is not a loop and E is removed from L .

Fig. 4. Loop filtering algorithm.

CBs using a two-phase algorithm. The first phase identifies all *repeated CBs* as loop candidates. The second phase filters those repeated **CBs** that implement loops from regular **CBs** that are just repeated a number of times. The latter **CBs** are then switched back to normal conditional branches. In the first phase, the CET profiler creates a list for each encountered **CB** instruction (identified by its address) and stores the branch's results for that **CB** instruction. Similar consecutive branch results (taken/not taken) are stored in one list's node, which has two fields: the result of the branch (T/NT) and a counter that represents how many consecutive times the branch was taken or not taken. As such, each loop candidate **CB** instruction will have a branch's results chain, as shown in Figure 3; for example, an inner loop with a constant count of 10 that is inside an outer loop of count 10 will have a chain of 10 T blocks, each having a count value of 10, and 10 NT blocks, each having a count of 1. Hence, an x86 conditional branch instruction is considered as a loop candidate under the following conditions:

1. All not-taken nodes have a counter of one.
2. The last node must be a not-taken node.
3. The first node can be either taken or not taken depending on if the loop is outer or inner. Thus, the loop instruction has a flag bit to indicate if the first node is taken or not. The loop body is the set of instructions between the **CB** and the target address. Referring to Table 1, repeated loops (i.e., inner loops) will be encoded either as LOOP (with constant counter), LOOP-C (with a counter that increments/decrements by a constant amount), or LOOP-R (with random counters). This algorithm will detect all properly nested loops with forward or backward edges. It may also consider some conditional branches as loops that were not intended to be loops; for example, if a conditional branch is executed twice, being taken the first time and not taken the second time, it will be considered as a loop with one iteration. This behavior, however, is still correct. In addition, in case some improperly nested loops are not recognized as loops, they will be regular **CBs**, but the contiguous data addresses and instruction addresses within these loops will still be compressed. The only overhead is the inclusion of their associated T/NT bits in the CET data, which is not very significant.

Figure 4 shows the second phase of the loop detection algorithm, which filters out the loop candidates during the CET emulation phase when all CET code is traversed. It employs a list of all loop candidates L , where each entry is the CET code of an identified candidate loop. During emulation, when a loop CET instruction is encountered for the first time, it is pushed to the front

Fig. 5. Using the CET thread synchronization primitive instructions for different synchronization events. Thread creation in simulation would follow the exact order as in the original execution (i.e., during the CET generation). For other synchronization events, however, thread arrival order may differ from the original order as illustrated in (b), (c), and (d), where threads are ordered from top to bottom according to their original arrival order, but the diagrams show their actual arrival order.

of L . Otherwise, if it is at the front of L , its counter is decremented (and popped out when its counter reaches zero). If the loop candidate is found in L but not at the front, this means it was just a **CB** that does not represent a loop, so it is removed from L and reverted to a regular **CB** CET instruction.

3.4 Thread Synchronization Events

Detecting thread synchronization events is straightforward with PIN instrumentation. PIN has what is called RNT object (routine object), which has many functions through which Pin can intercept different routines and function calls (including multithreading events, such as locks, unlocks, barriers, etc.). Hence, routines are intercepted by their names. Furthermore, PIN provides call-backs when each thread starts and ends. These are used by our tool to create the CET code and data files for each thread. As shown in Table 1, only four CET instructions (START, PAUSE, WAKE, and TERMINATE) are used to encode thread synchronization events such as synchronization barrier, access to critical sections (Locks/Unlocks), and atomic (read-modify-write) operations. Figure 5 illustrates that a simulator can simulate different synchronization events encoded with these instructions only (with no timestamps in the CET trace). Thread creation is straightforward with START instruction as shown in Figure 5(a). Thread synchronization at barriers can actually be modeled in two ways:

1. One that allows replicating the original thread arrival order at the barrier; see Figure 5(b). A WAKE (W) instruction is inserted (at the point of arrival) in the latest thread that arrives at the barrier during trace generation. PAUSE (P) instructions are inserted at the arrival points of the other threads. The simulator in this case would pause all threads until all of them arrive at the barrier and then reset each thread's time counter to that of the thread with the W instruction.

2. Another that allows a different thread arrival order (e.g., when simulating a different HW configuration than the one used for CET generation); see [Figure 5\(c\)](#). Only **P** instructions are inserted at the arrival points of all threads. In this case, the simulator would pause each thread that reaches a **P** instruction, until all threads are paused, and then all threads are resumed from the time counter of the last thread that reached the **P** instruction.

Similarly, serialization of threads' access to a critical section can also be handled in two ways:

1. One that allows replicating the original thread access order to the critical section. All threads will have **P** instructions (before the critical section) and a **W** instruction after the critical section, except for the first thread to arrive at the critical section (during CET generation), which will only have a **W** instruction after the critical section (as shown in [Figure 5\(d\)](#)). The simulator would pause each thread until its predecessor reaches the **W** instruction, and then that thread is granted access to the critical section and its time counter is set to the "**W** point" of the predecessor thread.
2. In the second method, all threads will have **P** instructions before the critical sections and **W** instructions afterward. The simulator in this case would use a "**P**" FIFO to pipeline thread requests to access the critical section. That is, if a thread reaches a **P**, it is queued in the "**P**" FIFO. Threads are then granted access to the critical section on a first-come-first-served basis. When the current thread reaches the **W** instruction, it is de-queued from the "**P**" FIFO and the next thread in line accesses the critical section and so on. Hence, the time difference between queuing a thread and granting it access represents the actual simulated spinning time of that thread.

Atomic operations (read-modify-write) are broken into LOAD, modify (one or more instructions such as ALU instructions), and STORE instructions. This STORE instruction is given a special CET code that distinguishes it from a regular store instruction so it can be used to trigger a simulator's coherency protocol.

3.5 CET Code Generator

In this phase, the profiled image is refined and its instructions are replaced by the corresponding CET instructions' formats. This includes processing the information regarding each instruction collected during the profiling phase. For example, in the case of a branch instruction, the loop detection algorithm is applied to check whether the instruction might be a loop or not. In the case of unconditional jump, it is checked whether this instruction jumps always to the same target or not and hence assigns the appropriate opcode for it. In case of load/store, it is checked whether it accesses a contiguous or noncontiguous block of data. The check for load/store can be performed earlier at the profiling phase (e.g., after the instruction references 50 or any specified number of memory addresses). This saves time and space. [Figure 6](#) shows the CET code generation algorithm.

To illustrate the proposed trace compression methodology, [Figure 7](#) shows a generated CET trace for a small C-code snippet (a loop to find the maximum of a 1-million-integer array). For this simple example, the compression ratio is approximately one-millionth (i.e., 0.000001).

3.6 Emulator and CET Data Generator

This phase generates the CET data in files with proper sequential order (i.e., a FIFO). Hence, when the CET code is processed (e.g., by a timing simulator), data required by any CET instruction can be consumed from the corresponding CET data file sequentially in the proper order in which they are needed. The other purpose of this step is to test the correctness of the CET code and report

1. For each thread *i*, let **CetCode** be an array of CET instructions.
2. For each instruction *I* in the profiled image of thread *i*, do the following:
3. If *I* is a conditional branch instruction:
  - a. If *I* is a loop candidate, mark it as a loop candidate
  - b. Else, if it is always taken, *CetCode.add(JUMP displacement)*
  - c. Else, if it is always not taken, *CetCode.add(ALU-INT)*
  - d. Else, *CetCode.add(BRANCH displacement)*
4. If *I* is a loop or loop candidate and its counters:
  - a. constant, *CetCode.add(LOOP displacement, counter)*
  - b. have constant difference *CetCode.add(LOOP-C disp., counter, inc.)*
  - c. have non-constant difference, *CetCode.add(LOOP-R disp.)*
5. If *I* is load/store and its memory addresses:
  - a. are the same, *CetCode.add(LOAD/STORE address)*
  - b. have constant nonzero difference, *CetCode.add(LOAD-C/STORE-C address, in.)*
  - c. have non-constant difference : *CetCode.add(LOAD-NC/STORE-NC)*
6. If *I* is an unconditional jump, return or call instruction:
  - a. If it has only one target address: *CetCode.add(JUMP displacement)*
  - b. If it has multiple target addresses: *CetCode.add(JUMP-M)*
7. Otherwise, add *I* into **CetCode** as it is.
8. Dump **CetCode** into a text file in binary format

Fig. 6. CET code generation algorithm.

Original program snippet ...	Generated CET Code mnemonics...
<pre>int main () { int array[1000000]; int max = array[0]; for(int i = 0; i < 1000000; i++) if(array[i] > max) max = array[i]; return 0; }</pre>	<pre>LOAD 140724540834992, Rd, Ra; STORE 140724540834984, Ra, Rb; STORE 140724540834988, Ra, Rb; JUMP 11, 1, Ra; LOAD 140724540834988, Rd, Ra; ALU Rd, Ra, Rb; LOAD-C 140724540834992, 4, Rd, Ra; LOAD 140724540834984, Rd, Ra; BRANCH 5, 1; LOAD 140724540834988, Rd, Ra; ALU Rd, Ra, Rb; LOAD-NC, Rd, Ra; STORE 140724540834984, Ra, Rb; LOAD 140724540834988, Rd, Ra; LOAD 140724540834988, Rd, Ra; Loop 11, 1000000, 0; ALU Rd, Ra, Rb; LOAD 140724544834992, Rd, Ra;</pre>

This segment appears a million times in the original trace

Fig. 7. Compression results for a simple C-code snippet.

bugs, if any. It should be noted that this step practically eliminates any need to decompress or reconstruct the original trace. Figure 8 shows the emulation and CET data generation algorithm.

4 EXPERIMENTAL RESULTS

4.1 Experimental Setup

The proposed methodology was evaluated using a wide range of benchmarks that included a subset of Splash-2 (Ohara et al. 1995), PARSEC (2007), MediaBench I (1997), and SPEC CPU 2000 (Standard Performance Evaluation Corporation 2000). The generated CET code was compared to the best trace compression techniques (Chen et al. 2013; Milenkovic and Milenkovic 2007), as well as TCgen compressed traces. For the SPEC CPU 200 benchmarks, only the first and 51st billion instructions were used to be able to compare with SBC, as was done in Milenkovic and Milenkovic (2007). Table 2 lists the used benchmarks with their input sets. Experiments on all benchmarks except the

1. For each *thread i*
2. Let *CetFifo* be the corresponding CET data FIFO of *thread i*.
3. Let *pc* = the initial address of the *thread i*.
4. Let *CetCode* is the CET code memory of *thread i*.
5. While *CetCode* not finished, do the following
6. *I* = *CetCode*(*pc*)
7. If *I* is a loop candidate that did not pass, switch it to a conditional branch.
8. If *I* is any branch instruction (JUMP, JUMP_M, BRANCH, LOOP etc), *pc* = target address.
9. Else, *pc* = *pc* + 1
10. If *I* is LOAD-NC/STORE-NC:
  - a. *CetFifo*.enqueue(*I*.addresses.front).
  - b. *I*.addresses.dequeue.
11. Else, If *I* is BRANCH
  - a. *CetFifo*.enqueue(*I*.BranchResults.front).
  - b. *I*.BranchResults.dequeue.
12. Else, If *I* is JUMP_M
  - a. *CetFifo*.enqueue(*I*.addresses.front).
  - b. *I*.addresses.dequeue.
13. Else, If *I* is LOOP-R, and this is a new outer iteration:
  - a. *CetFifo*.enqueue(*I*.counters.front).
  - b. *I*.counters.dequeue.
14. Convert *CETFifo* to binary and output it.

Fig. 8. Emulation and CET data generation algorithm.

SPEC CPU 2000 were run on an Intel Xeon CPU E5-2680 machine (similar to the one used in [Chen et al. \(2013\)](#)). For the SPEC CPU 2000 benchmarks, experiments were run on a Pentium 4 machine similar to the one used in [Milenkovic and Milenkovic \(2007\)](#). The CET tool has been evaluated in two modes: (1) Instruction Addresses (IA)-only mode in which the baseline trace entry consists of the instruction along with its address (i.e., 32-bit instruction address, 32-bit instruction), and (2) full mode, in which the whole trace, instructions, instructions' addresses, and data addresses (if any) are compressed (i.e., 32-bit instruction address, 32-bit instruction [32-bit data address]). The IA-only mode was included for comparison with [Chen et al. \(2013\)](#). All 23 traces that were used to evaluate our CET tool were also compressed using TCgen.

Two metrics were used to evaluate the CET tool: (1) compression ratio, the size of the uncompressed execution trace over the size of the generated CET trace (CET code + data), and (2) generation/compression/decompression speed in MIPS (millions of an execution trace's instructions generated/compressed/decompressed per second). In IA mode, each trace element (executed instruction) in the uncompressed trace is 64 bit, (32 bits for the instruction address and 32 bits for the instruction itself), whereas it is 96 bits in the full mode; an extra 32 bits are added to represent the data address, if any. Though generation/compression speed are evaluated, they are less important than the compression ratio and decompression speed since an execution trace is generated/compressed only once but is used many times.

4.2 Accuracy Evaluation

The accuracy of the generated CET code has been validated by comparing the number of instructions executed natively (under PIN instrumentation) for several benchmarks against those

Table 2. Benchmarks and Their Input Sets

Benchmark	Input Set
swaptions (small)	16 swaptions, 5,000 simulations
swaptions (medium)	32 swaptions, 10,000 simulations
swaptions (large)	64 swaptions, 20,000 simulations
Blackscholes (small)	4,096 options
Blackscholes (medium)	16,384 options
Blackscholes (large)	65,536 options
bodytrack (small)	4 cameras, 1 frame, 1,000 particles, 5 annealing layers
bodytrack (medium)	4 cameras, 2 frames, 2,000 particles, 5 annealing layers
bodytrack (large)	4 cameras, 4 frames, 4,000 particles, 5 annealing layers
LU	512 × 512 matrix
FFT	256K points
Ocean	258 × 258 ocean
Radix	256K integers
Water-sp	512 molecules
Water-nsq	512 molecules
cjpeg	input_base_4CIF.ppm
g721decoder	clinton.g721
g721encoder	clinton.g721.pcm
pegwit_d/e	Default
164.gzip, 179.art, 176.gcc, 181.mcf, 186.crafty, 300.twolf, 183.equake, 175.vpr, and 256.bzip2	The first and 51st billion instructions of the reference input set as in Milenkovic and Milenkovic (2007)

emulated in our tool suite (i.e., the last step in our methodology). [Table 3](#) summarizes the accuracy results. The differences are broken down to total number of instructions, load instructions, store instructions, and other instructions (that do not include threading/synchronization instructions). All thread and synchronization events (thread creation/termination, synchronization barriers, locks/unlocks, etc.) were identical between native execution and CET emulation. The table shows the accuracy results for one thread and 16 threads for all benchmarks. The relative differences were calculated as $100\% \times (\text{native execution value} - \text{emulated CET value}) / \text{native execution value}$. These results show that there are no systematic differences (+/- differences), and the largest difference in total number of instructions was $\sim 0.01\%$ (in the 16-thread *Radix* benchmark). It should be noted that these differences are not necessarily errors. Upon careful examination of both original and CET instructions, we discovered that these differences are mainly due to two reasons: (1) the many ways x86 compilers encode loops and (2) the way our loop detection algorithm works. The loop detection algorithm sometimes makes a +1 or -1 error in the inner loop's iteration counters. Hence, benchmarks with more inner loops incur more errors. This is also evident from the fact that, for the same benchmark, as the number of threads increases, the errors also increase since loops are basically duplicated among threads and the loop counters' values are divided by the number of threads. The second reason is related to the way x86 instructions are translated to CET code. In order to make the CET code compatible with general load/store architectures, x86 memory-register instructions are translated to instruction pairs: Load-ALU or ALU-Store instructions, which increase the number of CET instructions relative to the original instructions.

Table 3. Differences Between the Number of Different Instructions in the Generated CET Trace and the Original Native Execution for Several Benchmarks for 1-Thread and 16-Thread Execution

Benchmark	1 Thread										16 Threads									
	Total Number of Instructions					Total Number of Loads					Total Number of Stores					Other Instructions*				
	Natively Executed	CET	Diff. %	Natively Executed	CET	Diff. %	Natively Exec.	CET	Diff. %	Natively Exec.	Natively Exec.	CET	Diff. %	Natively Exec.	CET	Diff. %	Natively Exec.	CET	Diff. %	Natively Exec.
swaptions	663549677	663549677	0.00000	204985515	204985515	0.00000	40073815	40073815	0.00000	418490347	418490347	418490347	0.00000	69884412	69884412	0.00000	232675422	232675422	-0.00137	261093362
Blackscholes	102274425	102274425	0.00000	21565922	21565922	0.00000	10824091	10824091	0.00000	29356648	29356648	29356648	0.00136	142246297	142246297	-0.00189	39207805	39207805	-0.00694	186918341
FFT	217280609	217282476	-0.00086	45677266	45677266	0.00091	69569247	69569247	0.00037	232672225	232672225	232672225	0.00067	261093362	261093362	-0.00135	186918341	186918341	0.01579	214738900
LU	445635004	445637211	-0.00050	143393532	143393532	0.00026	42604581	42604581	0.00121	261093362	261093362	261093362	0.00234	39207805	39207805	-0.00694	186918341	186918341	0.01579	214738900
Ocean	518596830	518599291	-0.00047	214902419	214901865	0.00119	16824582	16824582	0.00317	186918341	186918341	186918341	0.00317	214738900	214738900	0.01377	214738900	214738900	0.01377	214738900
Radix	93158275	93160160	-0.00202	37128609	37128167	0.00119	16824582	16824582	0.00317	186918341	186918341	186918341	0.00317	214738900	214738900	0.01377	214738900	214738900	0.01377	214738900
water.sp	272018359	272004768	0.00500	57713241	57730033	-0.02910	27385908	27385908	0.00317	186918341	186918341	186918341	0.00317	214738900	214738900	0.01377	214738900	214738900	0.01377	214738900
water.nsq	308049753	308036166	0.00441	63401694	63417578	-0.02663	29879688	29879688	0.00303	214738900	214738900	214738900	0.00303	214738900	214738900	0.01377	214738900	214738900	0.01377	214738900

*These do not include threading- and synchronization-related instructions (these are identical between native execution and CET emulation, for all benchmarks for all number of threads).

Fig. 9. Compression ratios of the generated CET traces for two modes; instruction addresses only (IA) and full modes for many benchmarks.

Fig. 10. Compression ratio versus problem size for three single-threaded benchmarks.

4.3 Compression Ratio

Figure 9 shows the compression ratio for the two modes. In general, IA mode has a higher compression ratio than the full mode, because IA mode ignores data memory references. Thus, the generated CET trace in IA mode does not include data addresses, which are often the largest component of the compressed trace. However, full mode can achieve a higher compression ratio when the application has few noncontiguous load/store addresses, such as ocean and blackscholes benchmarks. This is because the generated CET trace is nearly the same for the two modes, but the original execution trace is larger in the full mode. Figure 10 shows the compression ratio versus different problem sizes (small, medium, and large) of three different single-threaded benchmarks. From this figure, it is obvious that for the swaptions and blackscholes benchmarks, the compression ratio is nearly constant for the three aforementioned problem sizes. However, it decreases when the problem size is increased for the bodytrack benchmark. Increasing the problem size increases the uncompressed trace size. However, the effect of increasing the problem size on the generated CET trace size depends on the application structure, that is, the distribution of the noncontiguous addresses or dynamic unconditional jumps across the application. Thus, if the compressed trace size increases at the same rate as the uncompressed one, the compression ratio is sustained. Otherwise, the compression ratio might increase or decrease due to increasing

the problem size. The compression ratio achieved by the CET tool varies according to the application's structure, since it affects the content of the CET data. For example, a large number of noncontiguous memory addresses, dynamic function calls, dynamic unconditional jumps, large number of conditional branches inside loop bodies, and so forth result in larger CET data and therefore lower compression ratio, and vice versa.

Table 4 shows the original trace sizes, compressed trace sizes, and compression ratios achieved by the CET tool in comparison with the most notable compression methods in the literature: a compressor generated by TCgen, Ching-Wen Chen's (Chen et al. 2013), and SBC (Milenkovic and Milenkovic 2007). For the CET, results are shown for 23 single-threaded benchmarks for the two modes: full compression and instructions-only modes. Single-threaded benchmarks were used for comparisons with the other techniques' published results. Although the traces we used (x86 ISA) differ from Ching-Wen Chen's and SBC's, comparing the compressed trace sizes is still meaningful since these techniques do not retain the original trace addresses. As for the TCgen compressor, it was run on the same exact traces that were used with the CET tool. Our CET tool outperforms Ching-Wen's technique, which achieved a compression ratio between 32.3 and 47.6. Ching-Wen's technique has the same baseline trace as our IA mode. This table shows that the CET tool in IA mode achieved a better compression ratio than Chen's technique by at least one order of magnitude. Moreover, in the full mode, the CET tool is still better by at least one order of magnitude for most of the benchmarks. The CET tool does not have any case worse than Chen's technique. The CET tool in IA mode outperforms Chen's technique because it handles the instruction addresses in a different manner. Their compressed trace contains a very-long-bit vector, one bit per instruction, to indicate whether the current instruction's address is sequential or not. It also includes the differences among the nonsequential instruction addresses. On the other hand, the CET compressed trace captures the program flow control, and hence, when the CET code is executed, the instruction addresses are regenerated on the fly. CET compression also outperformed the TCgen compressor by orders of magnitude. The SBC compression ratios in Table 4 were obtained from Milenkovic and Milenkovic (2007) for two trace segments: first and 51st billion instruction segments. The authors also mentioned that the average segment size was 11GB; hence, the uncompressed traces averaged 22GB in total size, which is very close to our trace sizes (~22.4GB) for the same trace segments. The compressed trace sizes were estimated by dividing the average trace size by the reported compression ratio. While this is not accurate, it serves to estimate the order of the compressed trace size to have a meaningful comparison with CET.

Unlike the CET tool, SBC was not run on a fully detailed input trace but only a 38-bit Dinero (Edler and Hill 1998) memory trace. These traces were divided into two files each, one for instruction addresses and another for data addresses, and the 2-bit header stripped out (that identified whether the address was an instruction, data load, or data stored). This of course reduced the trace sizes by over 470MB of otherwise noncompressible data. It was not clear, however, how that would affect reconstruction of the original trace when decompressed. Most of the savings in the SBC method are from the use of variable-length stride and repetition count fields, and by separating the instruction and data addresses into two files and removing the 2-bit header from the original trace. The variable-length encoding complicates the uncompressing process, especially if it is done in an HW simulator, where alignment of the compressed trace entries in memory becomes an issue. Furthermore, the removal of the uncompressible 2-bit header and separating the instructions and data addresses into two files, although saving hundreds of megabytes, means that the original trace may not be reconstructed with the original order of memory references (instructions and data), which in turn limits the technique's usefulness in architectural simulations. As Table 3 shows, for all considered benchmarks, CET's compressed files have similar sizes as those

Table 4. Uncompressed and Compressed Traces Size and Compression Ratios for All Benchmarks and Comparisons to Other Work

Splash-2, PARSEC, & MediaBench Benchmarks	This Work (CET Compression)					TC Gen		Ching-Wen Chen, et. al. (IA Mode only) Compression Ratio		
	Uncomp. Trace Size (MB)	Comp. Trace Size (Full Mode) (MB)	Comp. Trace Size (IA Mode) (MB)	Comp. Ratio (Full Mode)	Comp. Ratio (IA Mode)	Comp. Trace Size (MB)	Comp. Ratio	Uncomp. Trace Size (MB)	Comp. Trace Size (MB)	Comp. Ratio
cjpeg	580.0	8.8287	0.288	66	1,343	243.5	2.4	459	14.2	32.3
g721decoder	1,649.2	4,7546	4.077	347	270	928.9	1.8	78	1.6	50
g721encoder	5,279.4	13,2014	11,037	400	319	3,072	1.7	98	3	32.3
pegwit_d	106.5	2,1782	0.266	49	267	58.5	1.8	747	15	50
pegwit_e	37.7	0.0489	0.043	771	584	8.6	4.4	1,331.2	26.6	50
FFT	2,486.6	4,6491	2,763	535	600	512.7	4.9	49,664	1,091.5	45.5
swaptions (large)	121,298.2	262,9626	131,405	461	615	58,368	2.1			
Blackscholes (large)	18,721.4	69,0176	69,018	271	181	6,656	2.8			
bodytrack (large)	153,523.0	1017.44	189,738	151	539	29,696	5.2			
LU	5,099.9	11,4785	11,463	444	297	1,638.4	3.1			
Ocean	5,934.9	2,7088	2,588	2,191	1,529	2,560	2.3			
Radix	1,066.1	6,0449	2,035	176	349	70.7	15.1			
water.sp	3,113.0	58,4571	5,772	53	360	1,331.2	2.3			
water.res	3,525.4	63,8028	6,091	55	386	1,433.6	2.5			
SPEC CPU 2000 Benchmarks						SBC [Milenkovic and Milenkovic 2007] [*]				
						Ave		Reported		
						Uncompressed Trace Size	Compression Ratios	Estimated Ave Compressed Trace Size (MB)		
164.bz2ip		463.32	33.97	49.4	449.3	~22 GB (11GB per segment)	70	321.8		
179.art		17.77	17.63	1288.1	865.3		77.5	290.7		
176.gcc		748.74	40.98	30.6	372.3		55	409.6		
181.mcf		272.70	44.42	83.9	343.5		38	592.8		
186.crafty		365.34	28.32	62.6	538.9		55	409.6		
300.twolf	22,888.2 (22.4 GB)	649.48	37.40	35.2	408.0		42	536.4		
183.equake		636.22	9.18	36.0	1661.8		55	409.6		
175.vpr		634.85	30.91	36.1	493.7					
256.bzip2		499.72	31.82	45.8	479.5					
Min	37.7	0.05	0.04	30.6	22.9					
Max	153,523	959.76	132.06	2,987.9	20,283.7					
Average	22,974.6	246.90	25.11	502.1	1,633.9					

^{*} Separated 38-bit instruction and data addresses and removed the 2-bit (uncompressible) headers (over 470MB of data). Uncompressed trace size was not reported for each benchmark, only the average segment size (11GB). Compressed trace size is estimated based on this average trace size and reported compression ratios.

Fig. 11. CET's compression ratio versus number of threads.

compressed by SBC. CET traces, however, not only have fixed-length fields but retain most of the original trace details (not just memory references) and do not require any further decompression.

Figure 11 shows the full mode compression ratio of nine benchmarks for different numbers of threads, namely, one, two, four, eight, and 16 threads. In this experiment, the total uncompressed and compressed traces' sizes are the summations of the uncompressed and compressed traces' sizes of all threads, respectively. In most cases, the compression ratio remains nearly constant as the number of threads increases. Because the application is distributed on the available threads, the total uncompressed and compressed traces' sizes do not change markedly. However, the compression ratio decreases for the ocean benchmark. This variation is due to the variation of the CET data size, especially the number of noncontiguous addresses, when the number of threads changes.

Sources of Compression

Table 5 shows a breakdown of the sizes of the CET code and different CET data components for several benchmarks in comparison with the original executable and data file sizes. It also lists the number of loops per benchmark and the percentage of these loops with constant or contiguous data addresses (i.e., addresses that do not need to be stored in the CET data files). In addition to compressing instruction addresses, loops with such data references result in more compression since no data references need to be saved. The different CET data types were explained in Section 3.1. It should also be noted that these benchmarks generate huge datasets from initial small data inputs. As this table shows, the CET code sizes are very small and most of the compressed trace size is in its data. These data are of course very important to recreate the original execution events and their size varies depending on the nature of the benchmark. CET addresses, conditional branch results, and jump displacements are the most prominent components in the CET data.

The results in Table 5 give insight into the workings of the proposed trace generation/compression methodology. Applications with regular loops that access the same data or regularly referenced data would achieve the highest compression. This is evident from the small sizes of the CET data files, except the branch result file. For example, for the *Ocean* benchmark, the branch results data file is relatively large, indicating a high number of conditional branching. Yet, the other three data files are small, indicating regular instructions/data references and hence the large compression ratio. On the other hand, benchmarks with large CET addresses and/or irregular branch target address (i.e., Jump-M) files achieved the lowest compression even if they contained many loops with regular data references (e.g.m *Water-sp*, *nsq*, and *Cjpeg* benchmarks).

Table 5. Breakdown of the Compressed Trace Component Sizes (All in MB) for Several Benchmarks in Comparison with the Original Executable and Data File Sizes

Benchmark	Exe Size	No. of Loops	% of Loops with Const./Contig. Data Refs	Input Data Size	Uncomp. Trace Size	CET Code size	CET Addresses Size	Branch Results Size	JUMP_M Displacements Size	LOOP_R Counters Size	Compressed Trace Size	Compression Ratio (Full Mode)
Cjpeg	0.095	236	93%	1.200	579,990	0.024	8.540	0.257	0.007	0.000	8.829	66
g721decoder	0.871	26	77%	0.072	2,956,960	0.032	0.678	3.071	0.516	0.458	4.755	347
g721encoder	0.871	26	74%	0.230	5,279,390	0.032	2.164	9.834	1.171	0.000	13.201	400
pegwit_d	0.946	48	77%	-	106,530	0.051	1.912	0.166	0.049	0.000	2.178	49
pegwit_e	0.946	29	83%	0.089	37,670	0.039	0.006	0.001	0.003	0.000	0.049	771
FFT	0.738	35	80%	4.000	2,486,580	0.037	1.887	0.721	2.005	0.000	4.649	535
swaptions (large)	3.800	25	88%	-	121,298,770	0.015	131.557	128.947	0.002	2.441	262.963	461
Blackscholes (large)	0.133	9	89%	0.063	18,721.380	0.008	0.000	6.258	62.752	0.000	69.018	271
Bodytrack (large)	4.500	107	91%	13.600	153,522.980	0.143	827.702	118.142	57.682	13.771	1,017.440	151
LU	0.748	35	100%	2.000	5,099,890	0.039	0.015	5.946	5.478	0.000	11.479	444
Ocean	0.798	247	99%	0.510	5,934,870	0.099	0.121	2.312	0.134	0.043	2.709	2,191
Radix	0.737	31	97%	0.001	1,066,110	0.034	4.010	0.001	2.000	0.000	6.045	176
Water-sp	0.762	91	88%	0.004	3,113,000	0.069	52.685	3.331	2.318	0.054	58.457	53
Water-nsq	0.761	86	86%	0.004	3,525,350	0.068	57.713	3.860	2.106	0.057	63.803	55

Fig. 12. Trace generation/compression times for CET and SBC.

Fig. 13. Decompression times for CET and SBC techniques.

4.4 Compression/Decompression Speed

Figure 12 and Figure 13 show the generation/compression and decompression times for the CET and SBC techniques. The CET tool was run as a single thread on a similar machine to the one reported in Milenkovic and Milenkovic (2007). It should be noted that for the CET, the decompression time is actually part of the trace generation time also (to produce ready-to-use CET code). Though SBC has a near-constant compression time, the CET tool is faster than SBC by orders of magnitude. This is due to the variable field-length encoding/decoding of the SBC technique, which requires manipulating large linked lists. The CET's decompression stage simply executes the CET code, and once a CET datum is required, it will be ready on the front of the corresponding FIFO, that is, no complex decoding steps. Though all benchmarks had the same number of instructions, the CET generation time varies depending on the nature of programs since it uses dynamic data structures (linked lists) during CET code and data generation.

Table 6 shows the CET trace generation speed in MIPS. It should be noted that, as with all trace compression techniques, the trace generation time is several orders of magnitude larger than the actual execution time of the original applications/benchmarks. The average trace generation speed is 186.4 MIPS, while the maximum is 789.1 MIPS (for the Bodytrack benchmark). These

Table 6. Generation/Decompression Speed (MIPS)

Benchmark	Compression Speed	Decompression Speed
Swaptions	623.5	10,599.2
blackscholes	62.9	65.4
Bodytrack	789.1	1,219.5
164.gzip	44.4	142.9
179.art	285.7	2,000.0
181.mcf	22.2	35.7
186.crafty	19.4	125.0
300.twolf	40.8	153.8
183.equake	57.1	2,000.0
175.vpr	46.5	333.3
256.bzip2	58.8	117.6
Min	19.4	35.7
Max	789.1	10,599.2
Average	186.4	1,526.6

Fig. 14. Normalized compression time versus number of threads.

results also show that decompression is much faster than generation. This is expected since decompression is simply decoding and executing the CET code. The generation speed depends on the benchmark's structure; for example, the longer the loop's chains and addresses' lists are, the slower the generation.

Figure 14 shows the compression time (normalized to that of a single thread) for five benchmarks as a function of the number of threads. As can be seen, compression time does not increase significantly with the number of threads. In fact, for some benchmarks, it decreases with the number of threads due to the smaller data structures that have to be handled. As mentioned earlier, the compression time depends on the total size of the application and more importantly the nature of the application.

5 CONCLUSIONS AND FUTURE WORK

In this work, a new method for generating compact traces of multithreaded applications is proposed. Such traces can be used in trace-based simulation of multicore architectures. The proposed on-the-fly generation method has several novelties over existing trace generation and compression

techniques, namely, retention of most execution event records, capturing multithreading-related synchronization events, no need for decompression (the generated trace can be directly interpreted/executed by a simulator), and compatibility with both SW- and HW-based simulators. A complete tool suite has been created to implement and evaluate the proposed method. On several benchmarks, experimental results showed that, compared to other methods, the proposed method can achieve a high compression ratio at a reasonable time while retaining good fidelity. Moreover, it had good scalability when the problem size and the number of threads were increased. Scalability can be further enhanced in the future by implementing interthread compression (extracting and compressing common areas of threads). This, however, has proven to be a challenge under the constraint of the no-decompression requirement.

REFERENCES

- S. B. (Intel). 2012. Pin - A dynamic binary instrumentation tool. Retrieved from <https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool>.
- E. Argollo, A. Falc, P. Faraboschi, M. Monchiero, and D. Ortega. 2009. COTSon: Infrastructure for full system simulation. *SIGOPS Oper. Syst. Rev.* 43, 1, 52–61.
- K. C. Barr and K. Asanovic. 2006. Branch trace compression for snapshot-based simulation. In *Proceedings of the 2006 IEEE International Symposium on Performance Analysis of Systems and Software*.
- K. C. Barr, H. Pan, M. Zhang, and K. Asanovic. 2005. Accelerating multiprocessor simulation with a memory timestamp record. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005 (ISPASS'05)*.
- S. Budanur, F. Mueller, and T. Gamblin. 2012. Memory trace compression and replay for SPMD systems using extended PRSDs. *Comput. J.* 55, 2, 206–217.
- M. Burtcher, I. Ganusov, S. J. Jackson, J. Ke, P. Ratanaworabhan, and N. B. Sam. 2005. The VPC trace-compression algorithms. *IEEE Trans. Comput.* 54, 11, 1329–1344.
- A. Butko, R. Garibotti, L. Ost, V. Lapotre, A. Gamatie, and G. Sassatelli. 2015. A trace-driven approach for fast and accurate simulation of manycore architectures. In *Proceedings of the 2015 20th Asia and South Pacific Design Automation Conference (ASP-DAC'15)*.
- T. E. Carlson, W. Heirman, and L. Eeckhout. 2011. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*.
- C.-W. Chen, C.-J. Ku, and T.-J. Liu. 2013. Efficient trace file compression design with locality and address difference. *J. Inf. Sci. Eng.* 29, 5, 1055–1070.
- J. Edler and M. D. Hill. 1998. Dinero IV Trace-Driven Uniprocessor Cache Simulator. Retrieved from <http://www.cs.wisc.edu/~markhill/DineroIV/>.
- E. N. Elnozahy. 1999. Address trace compression through loop detection and reduction. In *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*.
- A. Janapsatya, A. Ignjatovic, and J. Henkel. 2007. Instruction trace compression for rapid instruction cache simulation. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition, 2007 (DATE'07)*.
- E. Johnson, J. Ha, and M. B. Zaidi. 2001. Lossless trace compression. *IEEE Trans. Comput.* 14, 158–173.
- W. Jun, J. Beu, R. Bheda, T. Conte, D. Zhenjiang, and C. Kersey. 2014. Manifold: A parallel simulation framework for multicore systems. In *Proceedings of the 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'14)*.
- S. Kanev and R. Cohn. 2011. Portable trace compression through instruction interpretation. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*.
- A. Ketterlin and P. Clauss. 2008. Prediction and trace compression of data access addresses through nested loop recognition. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*.
- A. Khan, M. Vijayaraghavan, and S. Boyd-Wickizervind. 2012. Fast and cycle-accurate modeling of a multicore processor. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'12)*.
- Y. Luo and L. K. John. 2004. Locality-based online trace compression. *IEEE Trans. Comput.* 53, 723–731.
- J. Marathe, F. Mueller, T. Mohan, B. R. de Supinski, S. A. McKee, and A. Yoo. 2003. METRIC: Tracking down inefficiencies in the memory hierarchy via binary rewriting. Paper presented at the *International Symposium on Code Generation and Optimization, 2003 (CGO'03)*.
- MediaBench. 1997. <http://euler.slu.edu/~fritts/mediabench/>.
- A. Milenkovic and M. Milenkovic. 2003. Stream-based trace compression. *Comput. Arch. Lett.* 2, 4.

- A. Milenkovic and M. Milenkovic. 2007. An efficient single-pass trace compression technique utilizing instruction streams. *ACM Trans. Model. Comput. Simul.* 17, 1, 2.
- J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, and C. Celio. 2010. Graphite: A distributed parallel simulator for multicores. In *Proceedings of the 2010 IEEE 16th International Symposium on High Performance Computer Architecture (HPCA'10)*.
- S. Nilakantan, K. Sangaiah, A. More, G. Salvadory, B. Taskin, and M. Hempstead. 2015. Synchrotrace: Synchronization-aware architecture-agnostic traces for light-weight multicore simulation. In *Proceedings of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'15)*.
- M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. R. d. Supinski. 2009. Scalatrace: Scalable compression and replay of communication traces for high-performance computing. *J. Parallel Distrib. Comput.* 69, 8, 696–710.
- PARSEC. 2007. from <http://parsec.cs.princeton.edu/>.
- H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. 2010. PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*.
- R. Pengju, M. Lis, C. Myong Hyon, S. Keun Sup, C. W. Fletcher, and O. Khan. 2012. HORNET: A cycle-level multicore simulator. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.* 31, 6, 890–903.
- C. A. Prete, G. Prina, and L. Ricciardi. 1995. A trace-driven simulator for performance evaluation of cache-based multiprocessor systems. *IEEE Trans. Parallel Distrib. Syst.* 6, 9, 915–929.
- A. Rico, A. Duran, F. Cabarcas, Y. Etsion, A. Ramirez, and M. Valero. 2011. Trace-driven simulation of multithreaded applications. Paper presented at the *2011 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'11)*.
- Searchable Linux Syscall Table for x86 and x86_64. <https://filippo.io/linux-syscall-table/>.
- Standard Performance Evaluation Corporation. 2000. <https://www.spec.org/cpu2000/>.
- Z. Tan, A. Waterman, H. Cook, S. Bird, K. Asanovi, and D. Patterson. 2010. A case for FAME: FPGA architecture model execution. *SIGARCH Comput. Archit. News* 38, 3, 290–301.
- TCgen 2.0: A Tool to Automatically Generate Lossless Trace Compressors (2006).
- Valgrind Instrumentation Framework. 2000. <http://valgrind.org>.
- S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. 1995. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture, 1995*.
- M. Xu, R. Bodik, and M. D. Hill. 2003. A “flight data recorder” for enabling full-system multiprocessor deterministic replay. *SIGARCH Comput. Archit. News* 31, 2, 122–135.
- M. Xu, M. D. Hill, and R. Bodik. 2006. A regulated transitive reduction (RTR) for longer memory race recording. *SIGOPS Oper. Syst. Rev.* 40, 5, 49–60.
- J. J. Yi, R. Sendag, L. Eeckhout, A. Joshi, D. J. Lilja, and L. K. John. 2006. Evaluating benchmark subsetting approaches. In *Proceedings of the 2006 IEEE International Symposium on Workload Characterization*.
- X. Zhang and R. Gupta. 2005. Whole execution traces and their applications. *ACM Trans. Arch. Code Optim. (TACO)* 2, 3, 301–334.

Received October 2016; revised March 2017; accepted May 2017