Journal of Systems Architecture 61 (2015) 601–614

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

mcfTRaptor: Toward unobtrusive on-the-fly control-flow tracing in multicores

Amrish K. Tewar, Albert R. Myers, Aleksandar Milenković [*]

Department of Electrical and Computer Engineering, The University of Alabama in Huntsville, 301 Sparkman Drive, Huntsville, AL 35899, USA

ARTICLE INFO

Article history:
Received 30 September 2014
Received in revised form 17 April 2015
Accepted 6 July 2015
Available online 14 July 2015

Keywords:
Real-time embedded systems
Multicores
Software testing and debugging
Program tracing

ABSTRACT

Software testing and debugging has become the most critical aspect of the development of modern embedded systems, mainly driven by growing software and hardware complexity, increasing integration, and tightening time-to-market deadlines. Software developers increasingly rely on on-chip trace and debug infrastructure to locate software bugs faster. However, the existing infrastructure offers limited visibility or relies on hefty on-chip buffers and wide trace ports that significantly increase system cost. This paper introduces a new technique called mcfTRaptor for capturing and compressing functional and time-stamped control-flow traces on-the-fly in modern multicore systems. It relies on private on-chip predictor structures and corresponding software modules in the debugger to significantly reduce the number of events that needs to be streamed out of the target platform. Our experimental evaluation explores the effectiveness of mcfTRaptor as a function of the number of cores, encoding mechanisms, and predictor configurations. When compared to the Nexus-like control-flow tracing, mcfTRaptor reduces the trace port bandwidth in the range from 14 to 23.8 times for functional traces and 10.8–18.6 times for time-stamped traces.

1. Introduction

Embedded computer systems are essential in modern communication, transportation, manufacturing, medicine, entertainment, and national security. Faster, cheaper, smaller, more sophisticated, and more power-efficient embedded computer systems spur new applications that require very complex software stacks. The growing software and hardware complexity and tightening time-to-market deadlines make software development and debugging the most critical aspect of embedded system development. A recent study found that software developers spend between 50% and 75% of their time debugging programs [1], yet the nation still loses approximately \$20–\$60 billion a year due to software bugs and glitches. The recent shift toward multicore architectures makes software development and debugging even more challenging.

Ideally, software developers would like to have perfect visibility of the system state during program execution. However, achieving complete visibility of all internal signals in real time is not feasible due to limited I/O bandwidth, high internal complexity, and high operating frequencies. To address these challenges, modern embedded processors increasingly include on-chip debug infrastructure to capture, filter, buffer, and emit program and data traces. These traces, coupled with powerful software debuggers, enable a faithful program replay that allows developers to locate and correct software bugs faster.

Fig. 1 illustrates a typical embedded multicore system-on-a-chip (SoC) with its on-chip and off-chip trace and debug infrastructure. The multicore SoC includes various components, such as multiple processor cores (Core0–Core3), a DSP core, and a DMA core, all connected through a system interconnect. Each component includes its own trace and debug logic (trace modules) that captures program execution traces of interest. Individual trace modules are connected through a trace and debug interconnect to on-chip trace buffers. On-chip buffers store program execution traces temporarily before they are read out through a trace port to an external trace probe. The external trace probe typically includes large trace buffers in orders of gigabytes and interfaces to the target platform's trace port and to the host workstation. The host workstation runs a software debugger that replays the program execution off-line by reading and processing the traces from the external probe and executing the program binary. This way, software developers can faithfully replay the program execution on the target platform and gain insights into behavior of the target system while it is running at full speed.

* Corresponding author.
E-mail addresses: akt0001@uah.edu (A.K. Tewar), myersar@uah.edu (A.R. Myers), milenka@uah.edu (A. Milenković).

http://dx.doi.org/10.1016/j.sysarc.2015.07.005
1383-7621/© 2015 Elsevier B.V. All rights reserved.

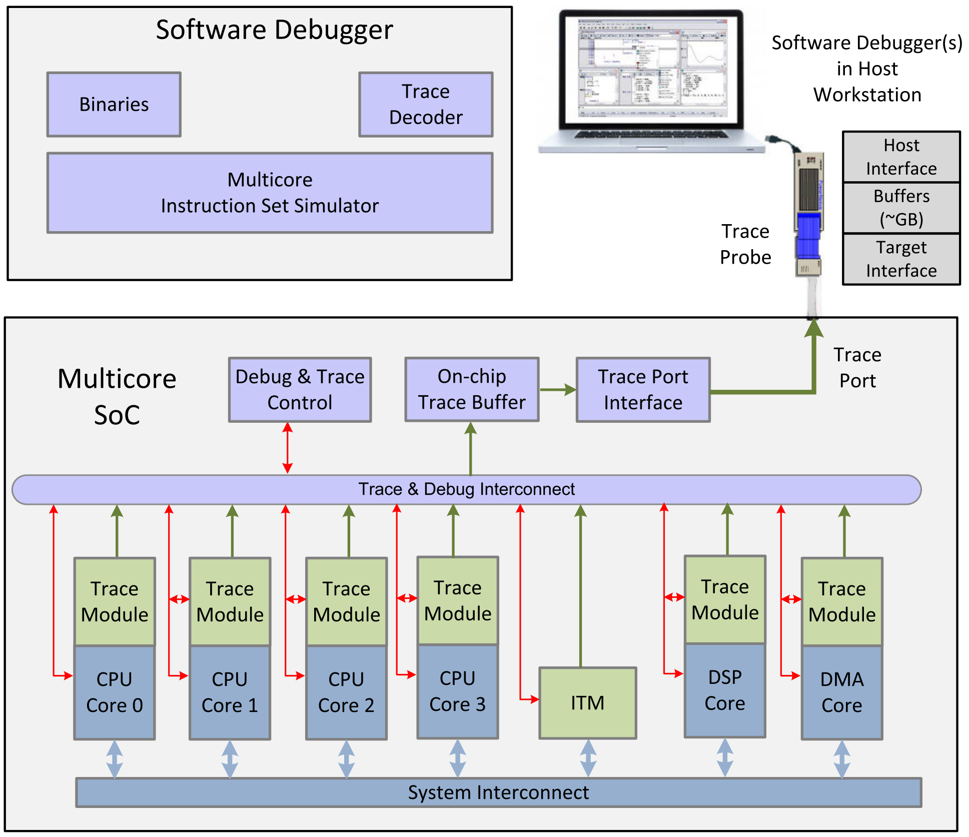


Fig. 1. Multicore SoC: debugging and tracing perspective.

Trace modules encompass logic that can support different classes of debugging and tracing operations. The IEEE's Nexus 5001 standard [2] defines functions and interfaces for debugging and tracing in embedded processors for four classes of debugging and tracing operations (Class 1–Class 4). Class 1 supports basic debug operations for run-control debugging such as single-stepping, setting breakpoints, and examining and modifying processor registers and memory locations when the processor is halted. It is traditionally supported through a JTAG interface [3]. The higher classes progressively add more sophisticated operations at the cost of additional on-chip resources (logic, buffers, and interconnects) solely devoted to tracing and debugging. Thus, Class 2 adds support for nearly unobtrusive capturing and streaming out of control-flow traces in real-time. Class 3 adds support for capturing and streaming out data-flow trace (memory and I/O read and write data values and addresses). Finally, Class 4 adds resources to support emulated memory and I/O accesses through the trace port.

Class 1 operations are routinely deployed in modern platforms. However, Class 1 operations are lacking in several important aspects as follows. First, they place burden on software developers to perform time-consuming and demanding steps such as setting breakpoints, single-stepping through programs and examining visually the content of registers and memory locations. Moreover, setting breakpoints is not practical or feasible in cyber-physical and real-time systems. Finally, since the processor needs to be halted, the debugging operations are obtrusive and may perturb sequence of events on the target platform and thus cause original bugs to disappear during debug runs. To address these challenges many chip vendors recently introduced trace modules with support for Class 2 and, less frequently, for Class 3 debug and trace operations. Some examples include ARM's CoreSight [4], MIPS's PDTrace [5], Infineon's MCDS [6], Freescale's MPC5500 Nexus implementation [7]. State-of-the-art trace modules employ filtering and encoding to reduce the number of bits necessary to recreate program execution. Yet, trace port bandwidths are still in the range of 1–4 bits per instruction executed per core for control-flow traces and 8–16 bits per instruction executed per core for data-flow traces [4]. With these trace port bandwidth requirements, a 1 KB on-chip buffer per processor core may capture control-flow of program segments in order of 2000–8000 instructions or data-flow of program segments of merely 400–800 instructions. Such short program segments are often insufficient for locating software errors in modern systems with more sophisticated software stacks where a distance between a bug's origin and its manifestation may span billions of executed instructions. Increasing the size of the buffers and the number of pins for trace ports is not an attractive alternative to chip manufacturers as it significantly increases the system complexity and cost. This problem is exacerbated in multicore processors where the number of I/O pins dedicated to trace ports cannot keep pace with the exponential growth of the number of processor cores on a single chip. Yet, debugging and tracing support in multicores is critical because of their increased proliferation in embedded systems and their increased sophistication and complexity.

Developing cost-effective hardware support for debugging and tracing in multicores is thus of great importance for future embedded systems. The on-chip debug and trace infrastructure should be able to unobtrusively capture control-flow and data-flow traces from multiple processor cores at minimal cost (which translates into minimal on-chip trace buffers) and stream them out in real-time through narrow trace ports.

Whereas commercially available trace modules typically implement only rudimentary forms of hardware filtering and compression with a relatively small compression ratio, several recent research efforts in academia and industry propose trace compression techniques that reach much higher compression ratios. Some of these techniques rely on hardware implementations of general-purpose compressors, such as LZ [8] or double-move-to-front [9]. Although they significantly reduce the size of the trace that needs to be streamed out, they have a relatively high complexity (50,000 gates and 24,600 gates, respectively). A set of recently developed techniques relies on architectural on-chip structures such as stream caches [10,11], and branch predictors [12,13] with their software counterparts in software debuggers, as well as effective trace encoding to significantly reduce the size of traces that needs to be streamed out. Thus, for control-flow traces Uzelac et al. [13] introduced TRaptor that achieves 0.029 bits per instruction on the trace port (∼34-fold improvement over Nexus-like trace) at hardware cost of approximately 5000 gates. ARM's new Embedded Trace Macrocell version four (ETMv4) proposed for ARMv7 ISA employs branch predictor to reduce the control-flow trace port bandwidth, reporting ∼0.35 bits per instruction [14]. For load value traces, Uzelac and Milenkovic [15,16] introduced cache first-access tracking mechanism (c-fiat) that reduces the trace size between 5.8 and 56 times, depending on the cache size. However, these techniques have been demonstrated for uniprocessors only. Hochberger and Weiss propose a new approach for debugging modern SoCs in real-time called hidICE [17]. hidICE relies on an emulator that replicates all master cores and memories from the target platform. It relies on the target platform to report only unexpected control-flows (exceptions) and data reads from peripherals that cannot be inferred by the emulator.

This paper focuses on capturing and compressing control-flow traces in multicores. They are an important tool in hardware and software debugging as well as in program profiling. Whereas certain classes of software bugs (e.g., data races) require data-flow traces, they need control-flow traces as well, e.g., to capture exceptions. Moreover, data-flow tracing is typically done only on a limited program segment rather than on the entire program because of the high costs. In such cases, control-flow traces and program check-pointing can be used to pinpoint the program segment for which a full data trace is needed. Capturing and compression of data traces in real-time is out of scope of this paper, but methods described by Uzelac and Milenkovic [16] can be extended to multicores.

We first analyze requirements for real-time control-flow tracing in multicores running a set of representative multithreaded benchmarks as a function of the number of cores ($N = 1$, 2, 4, and 8) (Section 2). We introduce a new hardware/software framework for capturing and compressing control flow traces in multicores called *mcfTRaptor* that builds on proven strengths of the existing TRaptor method [13] (Section 3). We analyze encoding of trace messages for both functional and time-stamped control-flow traces and evaluate effectiveness of variable encoding (Section 4). Section 5 describes our experimental environment. The experimental evaluation (Section 6) shows that *mcfTRaptor* dramatically reduces the number of trace messages thanks to high prediction rates in predictor structures. Thus, a multicore with eight cores streaming functional control-flow traces for a selected set of parallel benchmarks requires only 0.045 bits per instruction on the trace port with *mcfTRaptor* compared to 1.061 bits per instruction with Nexus-like tracing. Similarly, *mcfTRaptor* requires only 0.095 bits per instruction on the trace port compared to 1.759 bits per instructions when streaming time-stamped control-flow traces.

The main contributions of this work are as follows:

- We characterize trace port bandwidth requirements in multicore processors for Nexus-like functional and time-stamped control-flow traces. The trace port bandwidth is measured in bits per instruction executed and bits per clock cycle when executing a set of parallel benchmarks as a function of the number of processor cores.
- We introduce a technique called *mcfTRaptor* from *multicore control-flow tracing branch predictor* that exploits private branch-predictor like structures dedicated solely to program tracing to significantly reduce the number of trace messages.
- We evaluate a fixed and a variable encoding scheme for the events that are captured at *mcfTRaptor* structures.
- We perform a detailed experimental evaluation of the trace port bandwidth as a function of the *mcfTRaptor* configuration (*Small*, *Medium*, *Large*), the number of processor cores ($N = 1$, 2, 4. and 8), and encoding schemes. Our best performing configuration reduces the trace port bandwidth relative to the Nexus-like trace in the range of 14–23.8 times for functional control-flow traces and 10.7–18.6 times for time-stamped control-flow traces.
- In addition to analyzing the average trace port bandwidth, we analyze the trace port bandwidth dynamically during benchmarks' execution.

2. Control flow traces

Control-flow traces are created by recording memory addresses of all committed instructions in a program. However, such a trace includes a lot of redundant information that can be inferred by the software debugger with access to the program binary. To recreate the control-flow offline, the debugger needs only information about changes in the program flow caused by control-flow instructions or exceptions that cannot be inferred. When a change in the control-flow occurs, we could record the program counter (PC) and the branch target address (BTA) in case of a control-flow instruction or the exception target address (ETA) in case of an exception. However, such a sequence of (PC, BTA/ETA) pairs still contains redundant information. To reduce the number of bits to encode lengthy (PC, BTA/ETA) pairs, we can rather record the number of sequentially executed instructions in a dynamic basic block (also known as instruction stream) – a sequence of sequentially executed instructions starting at the target of a taken branch and ending with the first taken branch in the sequence. In addition, the target addresses of direct taken branches do not need to be recorded as they can be inferred by the software debugger. Therefore, only the following changes in the control-flow result in trace messages:

- A *taken conditional direct branch* generates a trace message that contains only the number of sequentially executed instruction in the instruction stream, also known as stream length, (SL, –);
- an *indirect unconditional branch* generates a trace message that includes the stream length and the address of the indirect branch, (SL, BTA); and
- an *exception event* generates a trace message that includes the message type, the number of instructions executed since the last reported event (*iCnt*), and the exception target address (ETA).

For multicores executing multithreaded programs, control-flow trace messages need to include information about the core on which a particular code segment has been executed. Without loss of generality, we assume that each thread executes on a single core ($Ti = Ci$); though threads can migrate between the cores, these migrations can be captured by system software rather than through hardware methods and can be merged with the hardware trace in the software debugger.

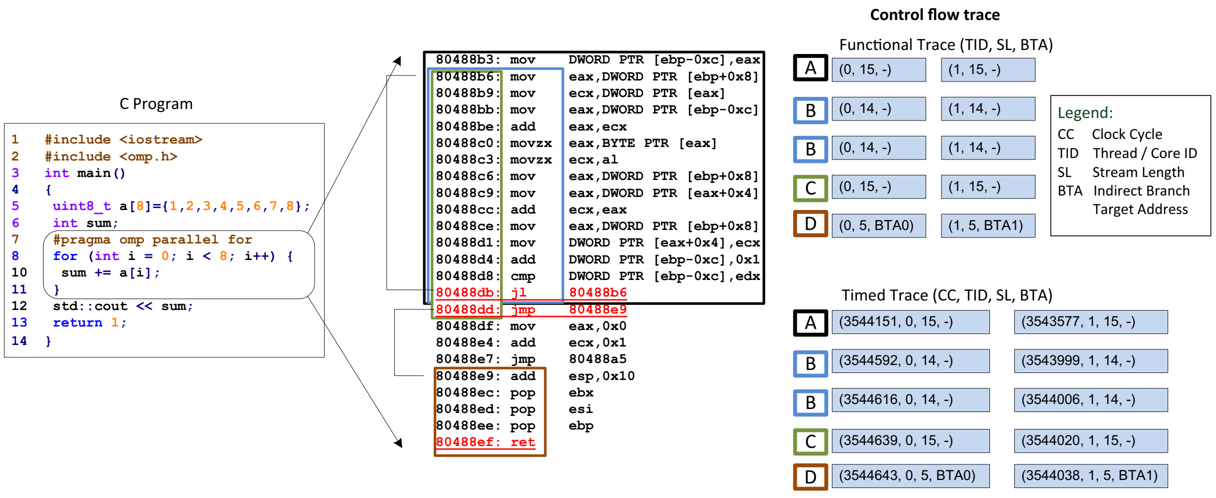
To illustrate tracing of a multithreaded program let us consider an OpenMP C program shown in Fig. 2 that sums up elements of an integer array. An assembly code snippet in the middle shows the instruction streams and the control-flow instructions in the inner loop: the stream A with 15 instructions starts at the address 0x80488b3, the stream B with 14 instruction starts at the address 0x80488b6, the stream C with 15 instructions starts at the 0x80488b6, and the stream D with 5 instructions starts at 0x80488e9. The same code snippet is executed in two threads ($Ti = 0$ and $Ti = 1$). The stream A is followed by two occurrences of the stream B, and one occurrence of stream C. Finally, the stream D ends with an indirect branch (*retq* instruction), so the last trace message will also include the target address (not known in compile time).

Fig. 2 top right shows the control-flow trace messages for both threads for the selected code segment. This trace does not show the ordering of the trace messages coming from different cores. However, the ordering of trace events is often useful in debugging parallel programs. We can assume that trace messages are time-stamped in trace modules using a global clock. These time stamps can then be used by a trace buffer control logic to properly order trace messages before they are streamed out of the chip. This way the software debugger would receive ordered control-flow messages. We refer to this kind of trace as functional control-flow trace – it is ordered, but it does not include time stamps. An alternative to this approach is to embed time stamps in trace messages and stream them out together with the core index, the stream length, and indirect branch target address. Fig. 2 bottom right shows the time-stamped control-flow trace for the inner loop. Each trace message carries information about the clock cycle in which the last instruction of a given instruction stream is committed. Time-stamped control-flow traces offer more insights to software developers and require simpler implementation of the trace buffer because trace messages can be streamed out as they arrive from the individual cores. However, time-stamped control-flow traces require higher trace port bandwidth because time stamps need to be streamed out, even though the time stamps can be differentially encoded. In this paper we will consider trace port bandwidth requirements for both functional and time-stamped control-flow traces.

In spite of generating trace messages only for events that cannot be inferred by the software debugger, the number of bits that need to be streamed out through the trace port remains relatively large. To illustrate the tracing challenges in multicores, we consider the minimal number of bits needed to capture control-flow traces when running representative benchmarks. Table 1 shows main characteristics of 10 benchmarks from the Splash2 benchmark suite compiled for the IA32 instruction set architecture (ISA) [18,19]. The benchmarks are run for a small input set for processor configurations with $N = 1$, 2, 4, and 8 processor cores. For each combination (*benchmark*, *N*), we show the number of executed instructions (IC – instruction count) and the number of instructions executed (committed) per clock cycle (IPC). The last row shows totals for the entire benchmark suite. Expectedly, the number of instructions slightly increases as the number of threads increases due to synchronization, especially for *cholesky* (from 1.27 with $N = 1$ to 2.77 billion of instructions with $N = 8$). The four columns in the middle show control flow instruction statistics for single-threaded programs, as they directly influence the frequency and size of trace messages that need to be streamed out. The column *Branch* shows the total frequency of control flow instructions, broken down to different types: conditional direct branches (C, D); unconditional direct branches (U, D); and unconditional indirect branches (U, I). Note: the IA32 ISA does not support conditional indirect branches. The benchmarks exhibit diverse characteristics, ranging from those with a relatively significant frequency of control flow instructions (e.g., *water_nsq*, *water_sp*, *lu*, and *raytrace*) to those with relatively low frequency (e.g., *radix*). The total frequency of control flow instructions in the instruction mix for all benchmarks is 10.46%. This relatively low frequency is due to Splash2 benchmark suite that includes predominantly scientific applications and computations kernels.

The benchmark's IPC as a function of the number of cores (*N*) indicates how well its performance scales. Thus, *water_nsq* scales very well because IPC ($N = 8$)/IPC ($N = 1$) = 7.55, but *radix* does not, because IPC ($N = 8$)/IPC ($N = 1$) = 3.41. The benchmark execution times in clock cycles are recorded using a cycle-accurate Multi2Sim simulator [20]. We model systems-on-a-chip with $N = 1$, 2, 4, and 8 cores. Each core has its private 8 KB data and instruction caches with hit latency of 2 clock cycles. The L2 cache is shared among cores and has latency of 4 clock cycles. The L2 cache size is set to $N \cdot 64$ KB. The main memory latency is set to 200 clock cycles.

To determine the required trace port bandwidth, we first extend Multi2Sim to capture program execution traces from multicore processor models. The collected time-stamped control-flow traces are post-processed to include the minimum necessary information to replay the program offline. The trace messages are encoded similarly to Nexus 5001 trace messages and include the fields described in Fig. 7a: thread/core identification (*Ti*), the number of


C Program
A
B
B
C
D
Control flow trace
Functional Trace (TID, SL, BTA)
Timed Trace (CC, TID, SL, BTA)
Legend:

```
#include <iostream>
#include <omp.h>
int main()
{
 uint8_t a[8]={1,2,3,4,5,6,7,8};
 int sum;
 #pragma omp parallel for
 for (int i = 0; i < 8; i++) {
  sum += a[i];
 }
 std::cout << sum;
 return 1;
}
```

```
80488b3: mov    DWORD PTR [ebp-0xc],eax
80488b6: mov    eax,DWORD PTR [ebp+0x8]
80488b9: mov    ecx,DWORD PTR [eax]
80488bb: mov    eax,DWORD PTR [ebp-0xc]
80488be: add    eax,ecx
80488c0: movzx  eax,BYTE PTR [eax]
80488c3: movzx  ecx,al
80488c6: mov    eax,DWORD PTR [ebp+0x8]
80488c9: mov    eax,DWORD PTR [eax+0x4]
80488cc: add    ecx,eax
80488ce: mov    eax,DWORD PTR [ebp+0x8]
80488d1: mov    DWORD PTR [eax+0x4],ecx
80488d4: add    DWORD PTR [ebp-0xc],0x1
80488d8: cmp    DWORD PTR [ebp-0xc],edx
80488db: jl     80488b6
80488dd: jmp    80488e9
80488df: mov    eax,0x0
80488e4: add    ecx,0x1
80488e7: jmp    80488a5
80488e9: add    esp,0x10
80488ec: pop    ebx
80488ed: pop    esi
80488ee: pop    ebp
80488ef: ret
```

Functional Trace (TID, SL, BTA)

A	(0, 15, -)	(1, 15, -)
B	(0, 14, -)	(1, 14, -)
B	(0, 14, -)	(1, 14, -)
C	(0, 15, -)	(1, 15, -)
D	(0, 5, BTA0)	(1, 5, BTA1)

Timed Trace (CC, TID, SL, BTA)

A	(3544151, 0, 15, -)	(3543577, 1, 15, -)
B	(3544592, 0, 14, -)	(3543999, 1, 14, -)
B	(3544616, 0, 14, -)	(3544006, 1, 14, -)
C	(3544639, 0, 15, -)	(3544020, 1, 15, -)
D	(3544643, 0, 5, BTA0)	(3544038, 1, 5, BTA1)

Legend:

CC	Clock Cycle
TID	Thread / Core ID
SL	Stream Length
BTA	Indirect Branch Target Address

Fig. 2. Capturing control-flow traces: an example.

Table 1
Splash2 benchmark suite characterization.

Benchmark	Instruction count [$\times 10^9$] (IC)				% Branches for $N = 1$				Instructions per cycle (IPC)			
Thread	$N = 1$	$N = 2$	$N = 4$	$N = 8$	Branch	C, D	U, D	U, I	$N = 1$	$N = 2$	$N = 4$	$N = 8$
barnes	2.13	2.13	2.13	2.14	10.25	6.10	2.44	1.71	0.41	0.83	1.52	2.76
cholesky	1.27	1.37	1.85	2.77	7.11	6.50	0.42	0.18	0.31	0.61	1.37	3.03
fft	0.92	0.92	0.92	0.92	8.35	6.06	1.14	1.14	0.28	0.56	1.00	1.58
fmm	2.79	2.79	2.84	2.88	7.05	6.57	0.36	0.12	0.40	0.80	1.57	2.94
lu	0.45	0.45	0.45	0.45	13.63	11.98	0.83	0.82	0.58	1.09	1.90	3.00
radiosity	2.23	2.32	2.31	2.31	11.70	6.48	3.40	1.81	0.64	1.24	2.35	4.42
radix	1.59	1.59	1.59	1.60	3.96	1.85	1.06	1.06	0.22	0.37	0.61	0.75
raytrace	2.47	2.47	2.47	2.47	12.12	7.74	2.65	1.73	0.50	1.17	2.16	3.60
water-nsq	0.74	0.74	0.74	0.75	14.12	11.56	2.16	0.41	0.70	1.46	3.04	5.29
water-sp	5.03	5.03	5.03	5.03	13.53	11.49	1.51	0.53	0.82	1.35	2.20	3.51
Total	19.61	19.81	20.33	21.31	10.46	7.82	1.69	0.95	0.45	0.87	1.56	2.57

sequentially executed instructions in a dynamic basic block (*SL*), and differentially encoded target address (*diffTA*) for indirect branches or exceptions. In case of time-stamped control-flow traces, trace messages include *diffCC* field. This field carries information about the number of clock cycles expired since the last reported trace event on a given processor core. The number of bits needed to encode the *Ti* field is $\lceil \log_2 N \rceil$ (0 for $N = 1$, 1 for $N = 2$, 2 for $N = 4$, and 3 for $N = 8$). To accommodate for variable length of the *SL*, *diffTA*, and *diffCC* fields, they are broken down into a certain number of chunks. Each chunk is followed by a so-called connect bit, *C*, which indicates whether the chunk is a terminating one ($C = 0$, end of the field) or not ($C = 1$, more chunks follow).

Table 2 shows the average trace port bandwidth for each benchmark for both the functional (NX_b) and the time-stamped (tNX_b) Nexus-like control-flow traces. The bandwidth is measured in the number of bits per instruction executed (*bpi*) and the number of bits per clock cycle (*bpc*). The total bandwidth (shown in row *Total*) in bits per instruction is calculated as the sum of trace sizes for all benchmarks divided by the sum of the number of instructions executed for all benchmarks. Similarly, the total bandwidth in bits per cycle is calculated as the sum of trace sizes for all benchmarks divided by the sum of the execution times in clock cycles for all benchmarks.

The required trace port bandwidth in bits per instruction increases as we increase the number of cores, due to additional information such as *Ti* that needs to be streamed out. The totals for functional traces range from 0.79 *bpi* for $N = 1$ to 1.06 *bpi* for $N = 8$. The required bandwidth varies across benchmarks and is highly correlated with the frequency and type of control-flow instructions. Thus, *radiosity* requires 1.06 when $N = 1$ and 1.34 when $N = 8$, whereas *fmm* requires only 0.41 when $N = 1$ and 0.65 when $N = 8$. In case of time-stamped traces the trace port bandwidth requirements increase significantly. Thus, the total bandwidth is 1.42 *bpi* when $N = 1$ and 1.76 *bpi* when $N = 8$. The individual benchmarks such as *lu*, *barnes*, and *radiosity* require significantly higher bandwidths. For example, *lu* requires 2.67 *bpi* when $N = 8$.

While the results for bandwidth in *bpi* show that the bandwidth requirements increase with an increase in the number of cores, they do not fully capture the pressure multiple processor cores place on the trace port, which is a shared resource. The trace port bandwidth in bits per clock cycle, *bpc*, better illustrates this pressure. Thus, *radiosity* with 8 threads executing on 8 cores requires 6.02 *bpc* on average for functional and 9.49 *bpc* for time-stamped control-flow traces. For $N = 8$ the total bandwidth for all benchmarks in *bpc* is 2.76 for functional and 4.58 for time-stamped traces. It should be noted that the bandwidth in *bpc* depends on processor and memory hierarchy model. Our model with IPC of 0.45 instructions committed in each clock cycle when $N = 1$ is a representative of embedded processors. More aggressive superscalar processor models will result in even higher trace port bandwidth requirements. These results indicate that capturing control-flow trace on the fly in multicores requires significantly higher trace port bandwidths and in turn larger trace buffers and wider trace ports. As shown in the next section, one alternative is to develop hardware techniques that significantly reduce the volume and size of trace messages that are streamed out.

3. mcfTRaptor

In this section we introduce a technique called *multicore control-flow Tracing Branch Predictor*, or *mcfTRaptor* for short. Fig. 3 illustrates a tracing and debug flow in multicores tailored for *mcfTRaptor*. Each processor core is coupled to its trace module through an interface that carries information about committed control-flow instructions. The trace module includes predictor

Table 2
Trace port bandwidth for functional and time-stamped Nexus-like (NX_b/tNX_b) control-flow traces for Splash2 benchmarks.

	Functional trace (NX_b)								Time-stamped trace (tNX_b)							
Benchmark	Trace port bandwidth [bpi]				Trace port bandwidth [bpc]				Trace port bandwidth [bpi]				Trace port bandwidth [bpc]			
Cores	$N = 1$	$N = 2$	$N = 4$	$N = 8$	$N = 1$	$N = 2$	$N = 4$	$N = 8$	$N = 1$	$N = 2$	$N = 4$	$N = 8$	$N = 1$	$N = 2$	$N = 4$	$N = 8$
barnes	0.98	1.06	1.13	1.21	0.40	0.87	1.72	3.35	1.67	1.74	1.82	1.90	0.68	1.44	2.76	5.28
cholesky	0.49	0.54	0.90	1.20	0.15	0.33	1.23	3.93	0.97	1.01	1.62	2.09	0.30	0.63	2.21	6.86
fft	0.80	0.87	0.94	1.00	0.22	0.49	0.94	1.59	1.44	1.50	1.56	1.62	0.40	0.84	1.56	2.57
fmm	0.41	0.46	0.55	0.65	0.17	0.37	0.87	1.93	0.81	0.87	1.00	1.13	0.33	0.69	1.56	3.35
lu	1.23	1.35	1.47	1.59	0.71	1.47	2.79	4.77	2.32	2.44	2.56	2.67	1.34	2.65	4.85	8.01
radiosity	1.06	1.15	1.25	1.34	0.68	1.43	2.92	6.02	1.81	1.90	2.01	2.11	1.15	2.36	4.69	9.49
radix	0.55	0.59	0.63	0.67	0.12	0.22	0.38	0.50	0.95	0.99	1.10	1.19	0.21	0.37	0.67	0.89
raytrace	1.05	1.13	1.21	1.30	0.53	1.32	2.63	4.68	1.80	1.88	1.96	2.05	0.90	2.19	4.24	7.37
water-ns	0.75	0.83	0.91	0.99	0.53	1.21	2.76	5.21	1.44	1.52	1.59	1.67	1.01	2.21	4.83	8.84
water-sp	0.78	0.86	0.93	1.01	0.64	1.15	2.05	3.54	1.47	1.54	1.62	1.69	1.20	2.08	3.57	5.95
Total	0.79	0.86	0.95	1.06	0.36	0.75	1.49	2.76	1.42	1.49	1.61	1.76	0.64	1.29	2.52	4.58

structures in hardware, solely dedicated to capturing and filtering control-flow traces. These structures are looked up when selected control-flow instructions commit in the corresponding processor core, namely conditional branches and indirect branches. For a given control-flow instruction, the predictor structures either (a) correctly predict its outcome or the target address; (b) incorrectly predict the outcome or the target address, or (c) cannot make prediction (e.g., due to a miss in a predictor). In all cases the predictor structures are updated based on their update policies, similarly to branch predictors in processor pipelines. The key insight that leads to significant reduction in the number and size of trace messages is that trace messages need to be generated only when rare mispredictions occur in the *mcfTRaptor* structures on the target platform. The messages are stored in a trace buffer, streamed out of the platform, and read by the software debugger. The software debugger has access to the program binary, instruction set simulator, and the trace messages captured on the target platform. It maintains software copies of all *mcfTRaptor* structures. These structures are updated during program replay in the same way their hardware counterparts are updated on the target platform.

Fig. 4 shows a block diagram of typical *mcfTRaptor* structures for a single core. The processor's trace module receives information about committed control flow instructions (PC, direct/indirect, taken/not taken, BTA) or exceptions (ETA) from the processor core and looks up the *mcfTRaptor* structures. *mcfTRaptor* includes structures for predicting (a) target addresses of indirect branches, e.g., an indirect Branch Target Buffer (iBTB) and a Return Address Stack (RAS) [21]; and (b) outcomes of conditional branches, such as outcome *gshare* predictor [22]. In addition, *mcfTRaptor* includes two counters: an instruction counter *Ti.iCnt* and a branch instruction counter *Ti.bCnt*. The *Ti.iCnt* is incremented upon retirement of each executed instruction, and the *Ti.bCnt* is incremented only upon retirement of control-flow instructions that could generate trace messages (e.g., conditional direct and unconditional indirect branches).

Fig. 5 describes operation of a trace module attached to core *i* when capturing control-flow tracing using *mcfTRaptor*. The instruction counter is incremented for each committed instruction. The branch counter is incremented for each control-flow instruction capable of generating a trace message. For indirect unconditional branches, the trace module generates a trace message only if the predicted target address does not match the actual target address. For direct conditional branches, the trace module generates a trace message only if the predicted outcome does not match the actual outcome. When a trace message is generated and placed in a trace buffer for streaming out, the counters *Ti.iCnt* and *Ti.bCnt* are cleared. The predictor structures are updated according to update policy. In case of exceptions, a trace message is generated with *Ti.bCnt* = 0 to indicate a special case, followed by the instruction count (*Ti.iCnt*) and the exception address (*Ti.ETA*).

The software debugger replays the instructions as shown in Fig. 6. The replay starts by reading trace messages for each thread and initializing the counters. If a non-exception trace message is processed, the software copy of *Ti.bCnt* is decremented. For indirect unconditional branches, if the counter reaches zero, the actual target address is retrieved from the current trace message; otherwise if (*Ti.bCnt* > 0), the target address is retrieved from the *mcfTRaptor* structures maintained by the software debugger. For direct conditional branches, if the counter reaches zero, the actual outcome is opposite to the one provided by the *mcfTRaptor* structures maintained by the software debugger; otherwise if (*Ti.bCnt* > 0), the actual outcome matches the predicted one. When *Ti.bCnt* reaches zero, the next trace message for that thread is read from trace probe. Handling of exception events is described in lines 3–8 in Fig. 6.

4. Trace messages encoding

mcfTRaptor reduces the number of trace messages that need to be reported, but the encoding scheme employed for these messages can also affect trace port bandwidth requirements. Trace messages should be encoded in such a way that minimizes the trace port bandwidth and reduces the hardware complexity of encoding and decoding trace messages. Table 3 summarizes *mcfTRaptor* events and necessary fields in trace messages for those events. All trace messages include a *Ti* field that carries information about the thread/core identification. The next mandatory field is the branch counter; for all non-exception fields this counter is greater than zero. Thus, the value zero is reserved to encode an exception event. The target address misprediction events also require the actual target address to be reported. In exception messages, the branch counter field is followed by the instruction counter (*Ti.iCnt*) and the exception target address (*Ti.ETA*). Note: these messages are tailored for IA32 ISA, and instruction sets with conditional indirect branches would require a slightly modified encoding scheme. In case of time-stamped control-flow traces, a time stamp field, *Ti.CC*, is included in each trace message.

One approach to encoding trace messages is to use fixed-length fields for encoding the counter values (*Ti.bCnt* and *Ti.iCnt*), and the target addresses (*Ti.BTA* and *Ti.ETA*). Whereas this approach simplifies encoding and decoding, it wastes the trace port bandwidth. The minimum number of bits to encode the counter values depends on frequency, type, and distribution of control-flow instructions in a traced program, as well as on prediction rates of

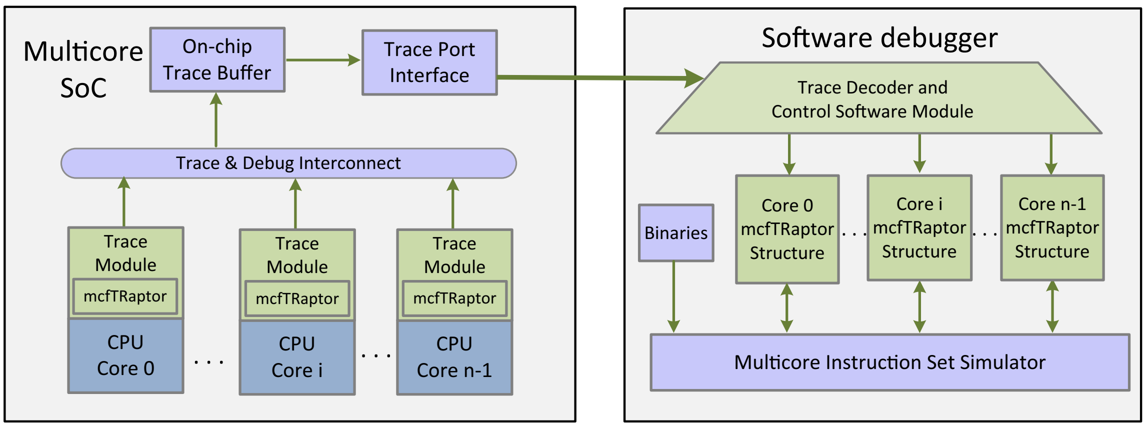

Multicore SoC
On-chip Trace Buffer
Trace Port Interface
Trace & Debug Interconnect
Trace Module
mcfTRaptor
CPU Core 0
Trace Module
mcfTRaptor
CPU Core i
Trace Module
mcfTRaptor
CPU Core n-1
Software debugger
Trace Decoder and Control Software Module
Binaries
Core 0 mcfTRaptor Structure
Core i mcfTRaptor Structure
Core n-1 mcfTRaptor Structure
Multicore Instruction Set Simulator

Fig. 3. Multicore SoC: system view of *mcfTRaptor*.

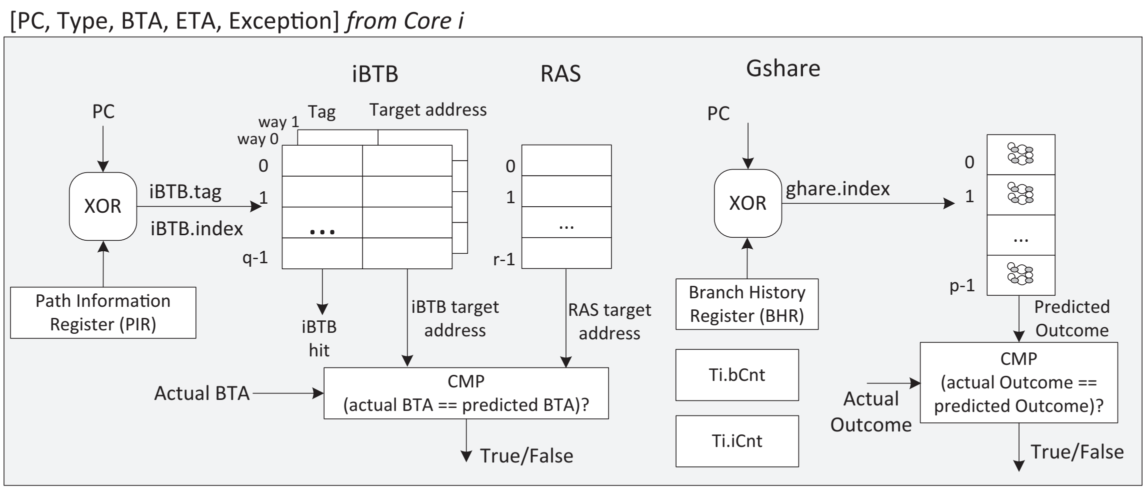


Fig. 4. *mcfTRaptor* structures for core *i*.

```
// For each committed instruction in Thread with index i on core i
Ti.iCnt++;  // increment iCnt
if ((Ti.iType == IndBr) || (Ti.iType == DirCB)) {
  Ti.bCnt++;  // increment bCnt
  // target address misprediction
  if ((Ti.iType == IndBr) && (Ti.BTA != p.BTA)) {
         Encode&Emit trace message <[Ti.CC], Ti, Ti.bCnt, Ti.BTA>;
         Place trace message into the Trace Buffer;
         Ti.iCnt = 0;
         Ti.bCnt = 0;
  }
  // outcome misprediction
  else if ((Ti.iType==DirCB) && (Ti.Outcome != p.Outcome)) {
         Encode&Emit trace message <Ti.CC, Ti, Ti.bCnt>;
         Place trace message into the Trace Buffer;
         Ti.iCnt = 0;
         Ti.bCnt = 0;
  }
  Update predictor structures;
}
if (Exception event) {
  Encode&Emit trace message for an exception event <Ti.CC, Ti, 0,
iCnt, ETA>;
  Place record into the Trace Buffer;
  Ti.iCnt = 0;
  Ti.bCnt = 0;
}
```

Fig. 5. *mcfTRaptor* operation on core *i*.

the *mcfTRaptor* structures. The prediction rates in turn depend on the predictors' size and organization.

An alternative approach is illustrated in Fig. 7b, called *TR_b*. The branch counter field is divided into 8-bit chunks. If an 8-bit field is sufficient to encode the counter value, the following field, called connect bit (*C*), has value zero, thus indicating the terminating chunk for *Ti.bCnt*. Otherwise, *C* = 1, and the following field carries information about the next 8 bits of the counter value. The trace messages for target address misprediction events carry information about the correct target address. An alternative to reporting the entire address (32-bit in our case) is to encode the difference between subsequent target addresses and thus exploit locality in programs to minimize the size of trace messages. The trace module maintains the previous target address, that is, the target address of the last mispredicted indirect branch (PTA). When a new target misprediction is detected, the trace module calculates the difference target address, *Ti.diffTA*, *Ti.diffTA* = *Ti.TA* – *Ti.PTA* and the *Ti.PTA* gets the value of current address *Ti.TA*, *Ti.PTA* = *Ti.TA*. The absolute value of *diffTA* is divided into chunks of 16-bits, and the connect bit indicates whether one or two 16-bit fields are needed to encode the message. The sign bit of the difference is encoded separately as shown in Fig. 7b.

The *Ti.CC* field can also be encoded with a fixed number of bits. Reporting the exact clock cycle in which a trace event has been captured is impractical because the program execution may take billions of clock cycles. Limiting the size of the *Ti.CC* field and

```
// For each instruction on core i
Replay the current instruction (if not trace event generating);
if (Exception message is processed) {
    Ti.iCnt--;
    if (Ti.iCnt == 0) {
      Go to exception handler at Ti.ETA;
      Get the next trace message;
    }
}
if ((Ti.iType == IndBr) || (Ti.iType == DirCB)) {
      Ti.bCnt--;  // decrement Ti.bCnt
      if ((Ti.iType == IndBr) && (Ti.bCnt > 0))
           Actual BTA = predicted BTA in software;
      else if (Ti.iType == DirCB) && (Ti.bCnt > 0))
           Actual outcome = predicted outcome in software;
      else if ((Ti.iType == IndBr) && (Ti.bCnt == 0))
           Actual BTA = BTA read from the trace message;
      else if (Ti.iType == DirCB) && (Ti.bCnt == 0))
           Outcome is opposite to predicted outcome;
      Update software predictor structures;
    if (Ti.bCnt == 0) Get the next trace message;
}
```

Fig. 6. Program replay in software debugger.

Table 3
mcfTRaptor trace messages.

mcfTRaptor events	Trace message fields
Outcome misprediction for direct conditional branch	<[Ti.CC], Ti, Ti.bCnt>
Target address misprediction for indirect unconditional branch	<[Ti.CC], Ti, Ti.bCnt, Ti.BTA>
Exception event	<[Ti.CC], Ti, 0, Ti.iCnt, Ti.ETA>

generating a periodic synchronization messages is a better approach, but would require periodic synchronization messages. An alternative is to apply differential encoding of time stamps. The trace module maintains the time stamp of the previous event (*Ti.PCC*). When a new trace event is detected, the module calculates the difference between the current and the previous time stamps, *Ti.diffCC* = *Ti.CC* – *Ti.PCC*. The *Ti.diffCC* is divided into chucks of 8 bits, and the connect bit is used to encode the entire field as shown in Fig. 7b. This approach for encoding time stamps is used for both Nexus-like trace messages (NX_b) shown in Fig. 7a and *mcfTRaptor* trace messages with fixed encoding (TR_b) shown in Fig. 7b.

By analyzing profiles of reported counter values (*Ti.bCnt*, *Ti.iCnt*, *Ti.diffCC*) as well as *diffTA* values, we determined that the number of required bits for encoding trace messages can be further minimized by allowing for variable encoding. Instead of using fixed-length chunks for *Ti.bCnt*, we allow for chunks of variable size, i_0, i_1, i_2, as shown in Fig. 7c. Similarly, we can use variable chunk sizes of lengths, j_0, j_1, j_2, for encoding *diffTA*, and h_0, h_1, h_2, ... for encoding *Ti.diffCC*. We call this encoding approach *TR_e*. The length of individual chunks is a design parameter and can be determined empirically. In determining the length of individual chunks, we need to balance the overhead caused by the connect bits and the number of bits wasted in individual chunks. A detailed analysis to find good chunk sizes is performed and selected parameters are used for all benchmarks. It should be noted that the variable encoding offers an additional level of flexibility to adjust encoding lengths for individual benchmarks or even inside different phases of a single benchmark. However, dynamic adaptation of the field lengths is left for future work.

5. Experimental environment

The goal of experimental evaluation is to determine the effectiveness of the proposed *mcfTRaptor* technique. As a measure of effectiveness, we use the average number of bits emitted on the trace port per instruction executed. As the workload we use control-flow traces of 10 benchmarks from the Splash2 benchmark suite [18,19] collected on a Multi2Sim simulator with IA32 ISA. In addition to evaluating the effectiveness of *mcfTRaptor*, the goal is to quantitatively assess the impact of *mcfTRaptor* configuration on its performance, as well as to find good chunk sizes (i_0, i_1, ..., j_0, j_1, ..., h_0, h_1, ...) and assess the effectiveness of variable encoding.

Fig. 8 shows the experimental flow used to create the control-flow traces and evaluate the trace port bandwidth. We have developed a custom extension to Multi2Sim called TmTrace. TmTrace captures time-stamped control-flow traces. The traces are generated for entire benchmark runs while varying the number of threads, $N = 1$, $N = 2$, $N = 4$, and $N = 8$. As a baseline case we use Nexus-like control-flow traces described in Section 2 with messages shown in Fig. 7a. The generated raw control-flow traces are forwarded to *mcfTRaptor* tool that models behavior of the mcfTRaptor predictor structures. The output *tNX_b* and *tTR_b* traces are read by the analyzer tool suite that performs encoding and analysis of statistics of interest for a selected set of encoding parameters which are input parameters for this stage. The statistics are maintained separately for each thread in a benchmark as well as the totals for the entire benchmark.

The trace port bandwidth depends on the following parameters: (a) benchmark characteristics – namely the type, frequency, and distribution of control-flow instructions, (b) the prediction rates of *mcfTRaptor* structures which in turn depend on their size and organization, and (c) the encoding parameters. To evaluate the impact of the *mcfTRaptor* predictors' size and organization, we consider three configurations called *Small*, *Medium*, and *Large*. Fig. 4 shows the *mcfTRaptor* predictor structures, including *gshare* outcome predictor (*p* entries), RAS (*r* entries), and iBTB (2-way with *q* sets). The index function for the outcome predictor is $gshare.index = BHR[log_2(p):0]\ xor\ PC[4 + log_2(p):4]$, where the BHR register holds the outcome history of the last $log_2(p)$ conditional

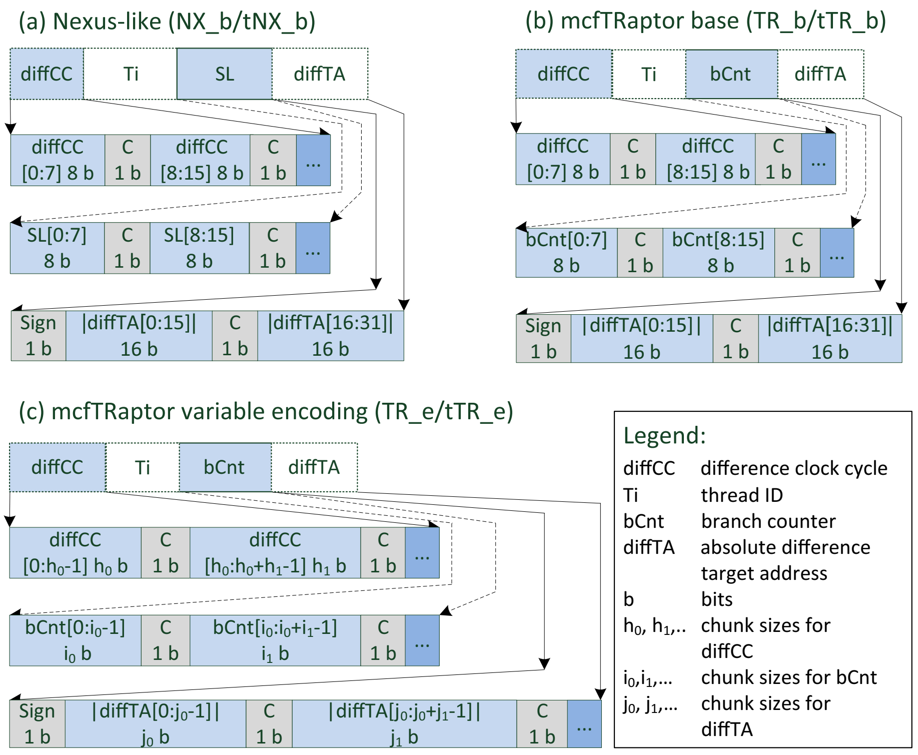


Fig. 7. Trace message encodings: (a) baseline Nexus-like (*NX_b*), (b) *mcfTRaptor* base (*TR_b*), and (c) *mcfTRaptor* variable encoding (*TR_e*).

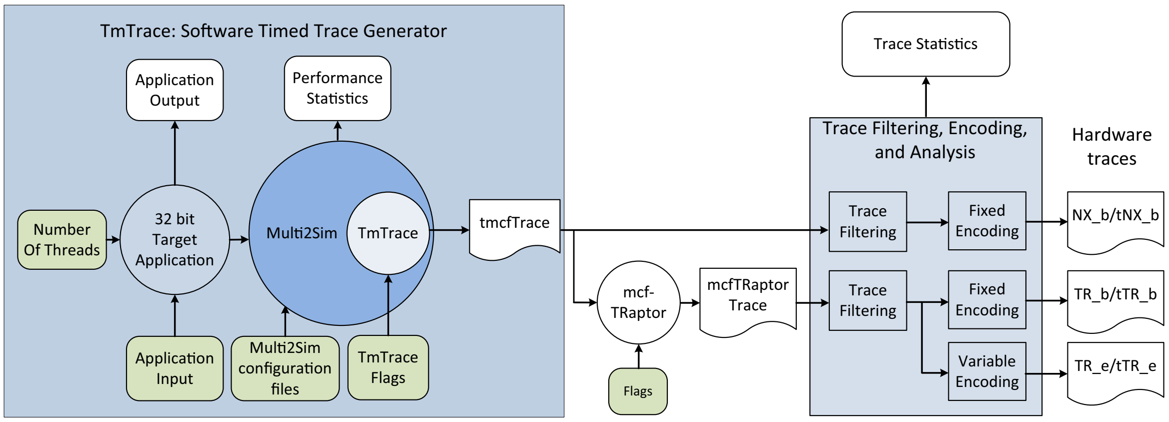


Fig. 8. Experimental flow.

branches. The iBTB holds target addresses that are tagged with an 8-bit tag. Both the *iBTB tag* and *iBTB index* are calculated based on the information maintained in the path information register (PIR) that is updated by control-flow instructions [23] [13]. For example, a BTB with 32 sets requires a 13-bit PIR that is updated as follows: $PIR[12:0] = ((PIR[12:0] \ll 2)\ xor\ PC[16:4])|Outcome$, where *Outcome* is the outcome bit of conditional branches. The iBTB tag and index are calculated as follows: $iBTB.tag = PIR[7:0]\ xor\ PC[17:10]$, and $iBTB.index = PIR[12:8]\ xor\ PC[8:4]$. The *Small* configuration includes a 512-entry *gshare* outcome predictor and an 8-entry RAS. The *Medium* configuration includes a 1024-entry *gshare* outcome predictor, a 16-entry RAS, and a 16-entry iBTB (2×8). The *Large* configuration includes a 4096-entry outcome predictor, a 32-entry RAS, and a 64-entry iBTB (2×32).

To evaluate the impact of encoding, we analyze trace port bandwidth for both encoding approaches TR_b and TR_e. To select good encoding parameters (i_0, i_1, j_0, j_1, ...), we profiled the Splash2 benchmarks to determine the minimum required bit length of the *Ti.bCnt*, *Ti.|diffTA|*, and *Ti.diffCC* fields. Fig. 9 shows the cumulative distribution function (CDF) for the minimum number of bits needed to encode *Ti.bCnt*, *Ti.|diffTA|*, and *Ti.diffCC* for the *raytrace* benchmark with $N = 2$ threads. This benchmark is selected because it has a relatively high frequency of control-flow instructions. The number of bits needed to encode the value of *Ti.bCnt* counters depends on benchmark characteristics as well as on misprediction rates of the *mcfTRaptor* predictors, which makes the selection of good parameters a challenging task. However, we can see that ~60% of possible *Ti.bCnt* values encountered during tracing *raytrace* can be encoded with 3 bits, and very few trace messages would require more than 8 bits. Similarly, over 70% of *Ti.|diffTA|* values encountered in the trace require less than 16 bits to encode. Whereas each (benchmark, *mcfTRaptor* configuration, *N*) triplet may yield an optimal combination of the encoding parameters, we search for a combination that would perform well across all benchmarks and all configurations. We limit the design space by requiring that $i_1 = i_2 = \ldots i_k$, and $j_1 = j_2 = \ldots = j_l$, where $\{i_0, i_1\} = \{1–6\}$ and $\{j_0, j_1\} = \{1–12\}$. We have found that the following values perform well for all configurations: $(i_0, i_1) = (3, 2)$, $(j_0, j_1) = (3, 4)$, and $(h_0, h_1) = (4, 2)$.

6. Results

The effectiveness of *mcfTRaptor* in reducing the trace port bandwidth directly depends on prediction rates as the trace messages are generated only on rare misprediction events. Table 4 shows the total misprediction rates collected on the entire benchmark suite for the *Small*, *Medium*, and *Large* predictor configurations, when the number of cores is varied between $N = 1$ and $N = 8$. The outcome misprediction rates decrease as we increase the size of the *gshare* predictor. For example, it is 7.86% for the *Small* configuration and 5.27% for the *Large* configuration when $N = 1$. It decreases with an increase in the number of processor cores as fewer branches compete for the same resources. Thus, the outcome misprediction rate is 6.23% for the *Small* configuration and 4.20% for the *Large* configuration when $N = 8$. The target address misprediction rates are also low because the RAS is correctly predicting targets of return instructions. The *Small* configuration does not include the iBTB predictor, resulting in misprediction rates ~8.5%. The *Medium* and *Large* configurations have only ~2% and ~0.6% target address mispredictions, respectively. These results demonstrate a strong potential of *mcfTRaptor* to reduce the trace port bandwidth requirements.

To quantify the effectiveness of *mcfTRaptor*, we first analyze the total trace port bandwidth for functional control-flow traces (Section 6.1) considering both *bpi* and *bpc* metrics. Next, we analyze the total trace port bandwidth for time-stamped control-flow traces (Section 6.2). Whereas the average trace port bandwidth allows us to quantify the effectiveness of the proposed techniques for filtering and encoding of trace messages, it does not fully capture the peak rates that occur in individual benchmarks during their execution. In Section 6.3 we look at the trace port bandwidth as a function of time during benchmark execution. In Section 6.4 we estimate hardware complexity of the proposed trace modules.

6.1. Trace port bandwidth analysis for functional traces

Fig. 10 shows the total average trace port bandwidth with the min–max ranges for the Nexus-like control flow traces (*NX_b*)

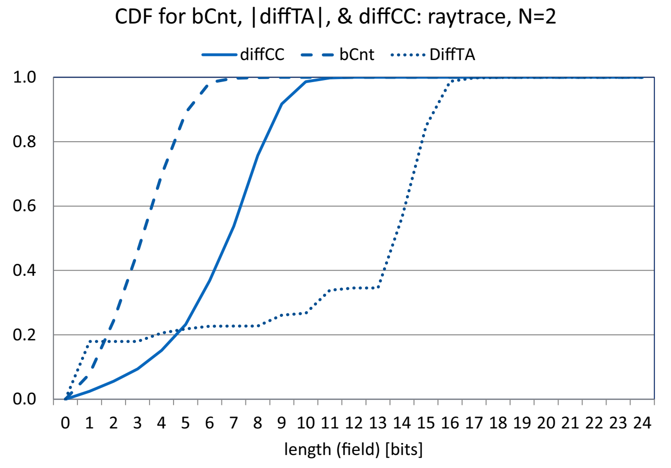


Fig. 9. Cumulative distribution function of the minimum length for bCnt, |diffTA|, and diffCC fields.

Table 4
Outcome and target address misprediction rates.

Configuration	Outcome misprediction rate [%]				Target address misprediction rate [%]			
	$N = 1$	$N = 2$	$N = 4$	$N = 8$	$N = 1$	$N = 2$	$N = 4$	$N = 8$
Small	7.86	7.83	7.20	6.23	8.47	8.59	8.58	8.55
Medium	6.69	6.67	6.14	5.31	2.05	2.12	2.08	2.05
Large	5.27	5.26	4.84	4.20	0.57	0.58	0.57	0.57

described in Section 2, the *mcfTRaptor* with base encoding (*TR_b*), and the *mcfTRaptor* with variable encoding (*TR_e*). For TR_b and TR_e we consider all three *mcfTRaptor* configurations (*Small*, *Medium*, and *Large*).

Table 5 shows the average trace port bandwidth in *bpi* for each benchmark separately when using the *Large* configuration for *TR_b* and *TR_e*. The results show that both *TR_b* and *TR_e* dramatically reduce the required total average trace port bandwidth relative to *NX_b*. The *NX_b* requires 0.79 *bpi* when $N = 1$ (ranging from 0.41 for *fmm* to 1.23 for *lu*) and 1.06 *bpi* when $N = 8$ (ranging from 0.65 for *fmm* to 1.59 for *lu*). TR_b is reducing the total average trace port bandwidth relative to *NX_b* in the range of 8.8 times ($N = 1$ with the *Small* configuration) to 18.6 times ($N = 8$ with the *Large* configuration). *TR_e* requires even lower trace port bandwidths. It reduces the total trace port bandwidth relative to *NX_b* in the range of 14.9 times ($N = 1$ with the *Small* configuration) to 23.8 times ($N = 8$ with the *Large* configuration). *TR_e* reduces the trace port bandwidth relative to *TR_b* in the range between 70% ($N = 1$) to 47% ($N = 8$) for the *Small* configuration, and 40% ($N = 1$) to 28% ($N = 8$) for the *Large* configuration. The benefits from variable encoding are higher in smaller configurations where higher misprediction rates lead to more frequent trace messages.

To further illustrate the strength of the proposed techniques, Fig. 11 shows the total average trace port bandwidth in bits per clock cycles for *NX_b*, *TR_b*, and *TR_e*. The experiments are run using processor models described in Section 2 resulting in IPCs shown in Table 1. The results show that *NX_b* requires 0.36 *bpc* (ranging from 0.12 for *radix* to 0.71 for *lu*) when $N = 1$ and 2.76 *bpc* (ranging from 0.5 for *radix* to 6.02 for *radiosity*) when $N = 8$. *TR_e* with the *Large* configuration requires only 0.015 *bpc* (ranging from ~10^{-6} for radix to 0.036 for *water-sp*) when $N = 1$ and 0.116 *bpc* (ranging from ~10^{-5} for *radix* to 0.352 for *water-ns*) when $N = 8$. *TR_e* requires on average less than 0.2 *bpc* in systems with $N = 8$ cores regardless of the number cores and the predictor configuration. More importantly, the critical benchmarks such as *radiosity* and *water-ns*, executing on a multicore with $N = 8$ cores with the *Large* configuration, still require less than 0.4 bits per clock cycle on the trace port (0.301 and 0.352 *bpc*, respectively).

6.2. Trace port bandwidth analysis for time-stamped traces

Fig. 12 shows the total trace port bandwidth in *bpi* for time-stamped control-flow trace *tNX_b*, *tTR_b*, and *tTR_e* as a function of the number of processor cores and predictor configurations. The total trace port bandwidth is broken down into individual fields of trace messages (DiffTA, bCnt, Ti, and diffCC). Expectedly, time-stamped control-flow traces increase the port bandwidth requirements. Table 6 shows the trace port bandwidth for all benchmarks for the *Large* configuration. For example, *tNX_b* requires 1.42 *bpi* when $N = 1$ (ranging from 0.81 for *fmm* to 2.32 for *lu*) and 1.76 *bpi* when $N = 8$ (ranging from 1.13 for *fmm* and 2.67 for *lu*). Both *tTR_b* and *tTR_e* significantly reduce the required trace port bandwidth. With the *Large* configuration TR_b requires 0.104 *bpi* when $N = 1$ (13.6 times improvement over *tNX_b*) and

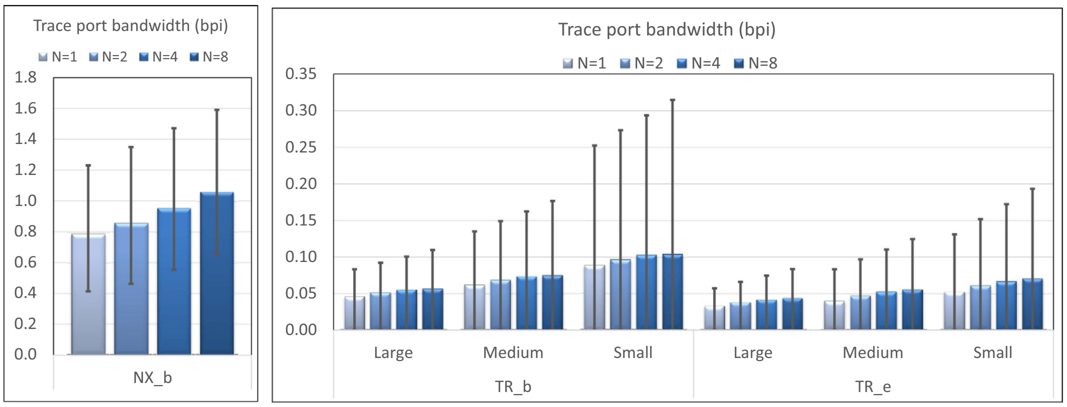


Fig. 10. Trace port bandwidth in *bpi* for functional control-flow traces.

Table 5
Trace port bandwidth analysis [*bpi*] for NX_b, TR_b (Large), and TR_e (Large).

# Threads/cores (N)	$N = 1$			$N = 2$			$N = 4$			$N = 8$		
Mechanism	NX_b	TR_b	TR_e	NX_b	TR_b	TR_e	NX_b	TR_b	TR_e	NX_b	TR_b	TR_e
barnes	0.981	0.041	0.028	1.056	0.045	0.033	1.133	0.050	0.037	1.210	0.054	0.042
cholesky	0.491	0.013	0.011	0.538	0.014	0.012	0.897	0.014	0.011	1.200	0.010	0.009
fft	0.803	0.007	0.007	0.869	0.008	0.008	0.935	0.009	0.008	1.002	0.010	0.009
fmm	0.413	0.021	0.018	0.463	0.023	0.020	0.554	0.025	0.022	0.650	0.027	0.024
lu	1.230	0.061	0.046	1.350	0.067	0.053	1.471	0.074	0.059	1.592	0.081	0.066
radiosity	1.064	0.069	0.045	1.151	0.078	0.054	1.250	0.087	0.062	1.336	0.090	0.066
radix	0.546	0.000	0.000	0.586	0.000	0.000	0.627	0.000	0.000	0.668	0.000	0.000
raytrace	1.052	0.083	0.057	1.134	0.092	0.066	1.214	0.100	0.074	1.299	0.110	0.083
water-ns	0.755	0.063	0.046	0.831	0.070	0.053	0.907	0.077	0.060	0.985	0.084	0.067
water-sp	0.781	0.063	0.044	0.857	0.070	0.051	0.933	0.077	0.058	1.008	0.084	0.065
Total	0.789	0.047	0.033	0.858	0.052	0.038	0.954	0.056	0.042	1.061	0.057	0.045

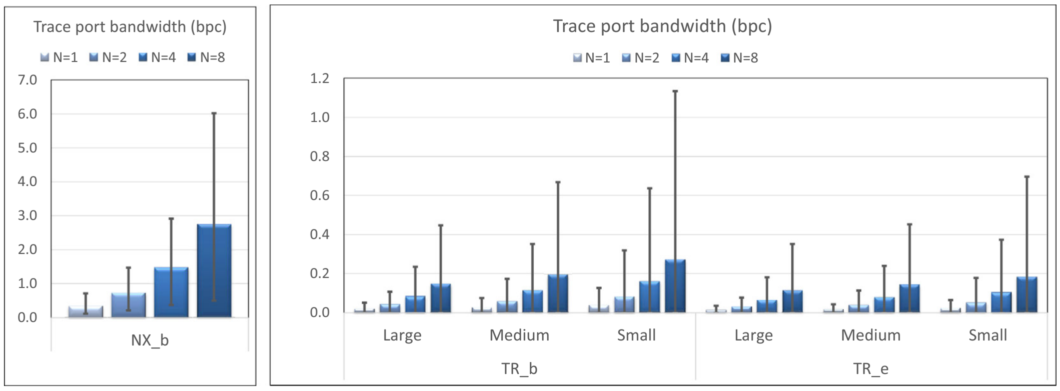


Fig. 11. Trace port bandwidth in *bpc* for functional control-flow traces.

0.111 when $N = 8$ (15.8 times improvement over *tNX_b*), whereas *TR_e* requires 0.087 *bpi* when $N = 1$ (16.3 times improvement over *tNX_b*) and 0.095 *bpi* when $N = 8$ (18.6 times improvement over *tNX_b*). *TR_e* with the *Large* configuration requires less than 0.1 *bpi* on the trace port regardless of the number of cores. The breakdown shows that the trace port bandwidth in *TR_e* becomes dominated by bits carrying time stamps.

Fig. 13 shows the total trace port bandwidth with the min–max ranges in *bpc* for *tNX_b*, *tTR_b*, and *tTR_e* as a function of the number of cores and *mcfTRaptor* configurations. The results show that *tNX_b* requires 0.64 *bpc* (ranging from 0.33 for *fmm* to 1.34 for *lu*) when $N = 1$ and 4.58 *bpc* (ranging from 0.89 for *radix* to 9.49 for *radiosity*) when $N = 8$. *tTR_e* with the *Large* configuration requires only 0.039 *bpc* (ranging from $\sim 10^{-6}$ for radix to 0.094 for *water-sp*) when $N = 1$ and 0.246 *bpc* (ranging from $\sim 10^{-5}$ for *radix* to 0.738 for *water-ns*) when $N = 8$. These results indicate that using time-stamped rather than functional control-flow traces more than doubles the required trace port bandwidth.

6.3. Dynamic trace port bandwidth analysis for time-stamped traces

Fig. 14 shows the trace port bandwidth as a function of time during execution of two benchmarks, *water-ns* and *radiosity*. The number of processors is $N = 8$. We analyze the bandwidth required for time-stamped control-flow traces for *tNX_b* and *tTR_e* for all three configurations, the *Large*, *tTR_e* (L), the *Medium*, *tTR_e* (M), and the *Small*, *tTR_e* (S). The benchmarks *water-ns* and *radiosity* are selected because they require the highest average total bandwidth for time-stamped control-flow traces. The trace port bandwidth in *bpc* is logged every 1 million clock cycles.

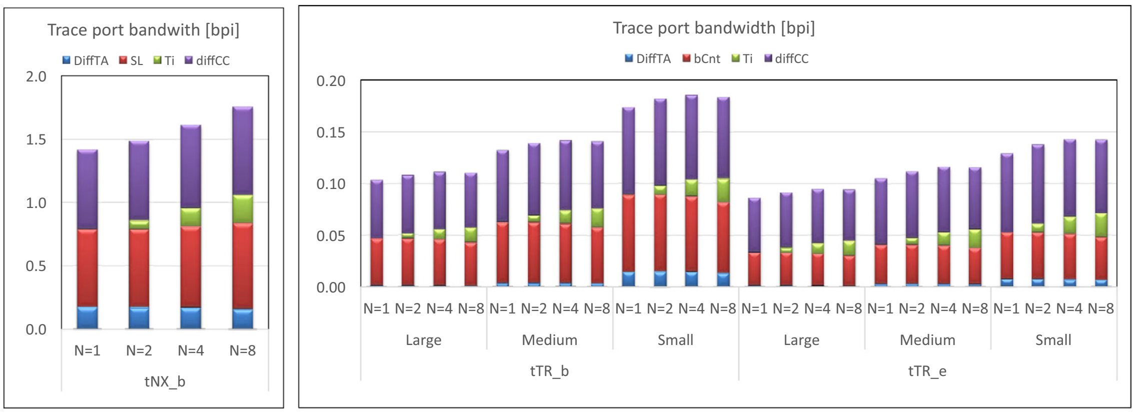


Fig. 12. Trace port bandwidth in *bpi* for time-stamped control-flow traces.

Table 6
Trace port bandwidth analysis [*bpi*] for *tNX_b*, *tTR_b* (Large), and *tTR_e* (Large).

# Threads/cores (*N*)	*N* = 1			*N* = 2			*N* = 4			*N* = 8		
Mechanism	tNX_b	tTR_b	tTR_e	tNX_b	tTR_b	tTR_e	tNX_b	tTR_b	tTR_e	tNX_b	tTR_b	tTR_e
barnes	1.672	0.099	0.079	1.738	0.105	0.084	1.820	0.110	0.089	1.904	0.115	0.094
cholesky	0.968	0.029	0.026	1.007	0.031	0.029	1.619	0.029	0.026	2.093	0.021	0.019
fft	1.438	0.018	0.017	1.499	0.018	0.018	1.563	0.020	0.019	1.625	0.021	0.020
fmm	0.810	0.057	0.051	0.866	0.059	0.054	0.995	0.061	0.055	1.129	0.062	0.057
lu	2.320	0.126	0.120	2.438	0.132	0.126	2.556	0.139	0.132	2.674	0.146	0.139
radiosity	1.805	0.151	0.116	1.896	0.161	0.125	2.008	0.170	0.134	2.106	0.170	0.136
radix	0.945	0.000	0.000	0.985	0.000	0.000	1.100	0.000	0.000	1.189	0.000	0.000
raytrace	1.802	0.182	0.147	1.877	0.189	0.155	1.957	0.198	0.163	2.048	0.211	0.174
water-ns	1.442	0.138	0.118	1.515	0.144	0.125	1.590	0.152	0.132	1.670	0.161	0.140
water-sp	1.467	0.134	0.115	1.543	0.140	0.122	1.619	0.147	0.129	1.695	0.155	0.137
Total	1.419	0.104	0.087	1.486	0.109	0.092	1.614	0.112	0.095	1.759	0.111	0.095

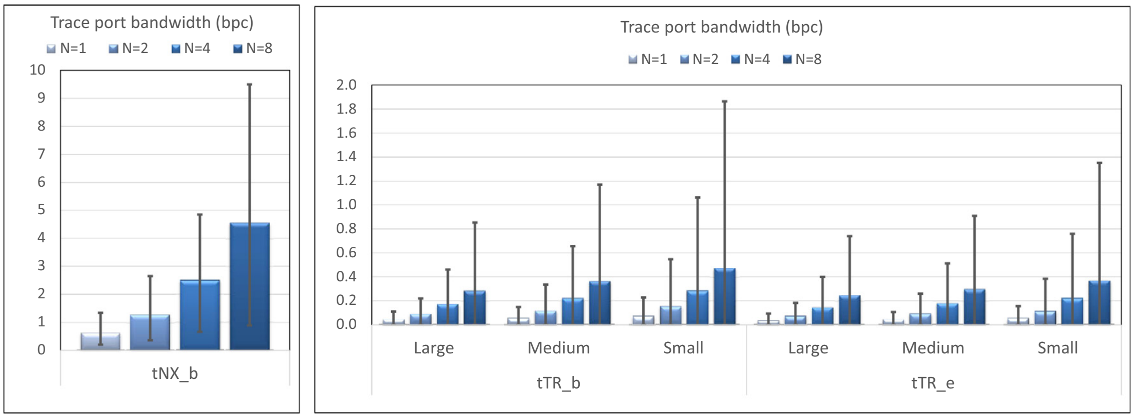
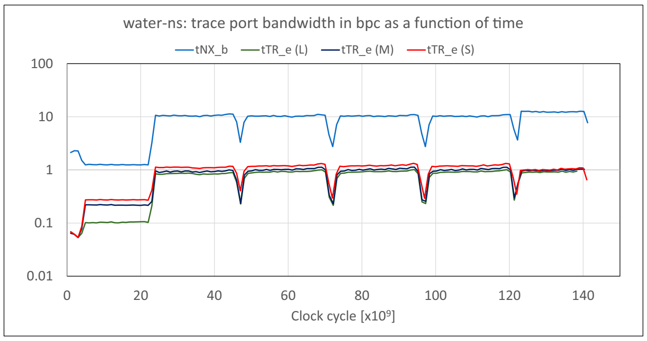


Fig. 13. Trace port bandwidth in *bpc* for time-stamped control-flow traces.

Let us first analyze *water-ns*. The average trace port bandwidth for *tNX_b* is 8.84 *bpc*, but the bandwidth varies over time from as low as 1.25 *bpc* to over 12.8 *bpc*. On the other side *tTR_e* (L) requires the average trace port bandwidth of 0.738 *bpc*. *tTR_e* (L) variations in the bandwidth mimic those in *tNX_b*, but they range between 0.05 *bpc* and 1.01 *bpc*. In case of *radiosity*, *tNX_b* requires the average bandwidth of 9.49 *bpc*. However, the bandwidth varies widely over time from as low as 0.28 *bpc* to 94.7 *bpc* at the very end of the program execution. Contrary to *tNX_b*, *tTR_e* requires 0.613 *bpc* on average, while varying between ~0 and 1.01 *bpc*. These results show that *mcfTRaptor* not only reduces the average trace port bandwidth, but also limits the variability of required bandwidth as the benchmarks move through different phases of program execution.

6.4. Hardware complexity estimation

To estimate the hardware complexity of the proposed trace modules, we estimate the size of all structures including the outcome gshare predictor, RAS, iBTB, registers PIR, PTA, PCC, as well as the estimated size of the output buffer. In estimating the fields that keep target addresses, we can eliminate the uppermost 12 address bits since they remain unchanged relative to the previous target addresses. The complexity of the *Small* configuration



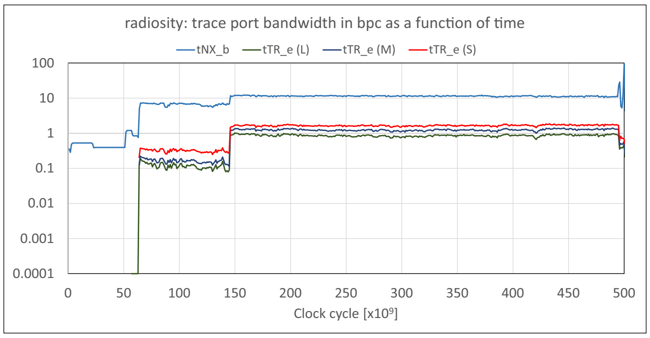


Fig. 14. Trace port bandwidth in *bpc* as a function of time during execution of *water-ns* and *radiosity* for $N = 8$.

(512-entry gshare and 8-entry RAS) is estimated to 2300 logic gates per processor core. The shared output trace buffer of $N * 64$ bits (where N is the number of processor cores) should be able to amortize all trace messages in case of time-stamped trace messages. The output trace buffer would need to be emptied at the rate of 2 *bpc* to accommodate streaming out time-stamped trace messages of *radiosity*. The *Medium* configuration is estimated to ~4800 logic gates per processor core. The output buffer would need to be emptied at the rate of 1.7 *bpc* if time-stamped trace messages are streamed out. Finally, the *Large* configuration requires ~16,500 logic gates. The output buffer needs to be emptied at the rate of ~1.3 *bpc* to accommodate the worst-case (*water-ns*). It should be noted that this analysis assumes tracing of time-stamped control-flow messages and the processor model described in Section 2. Our results indicate that it would be beneficial to ensure on-chip ordering of trace messages and streaming out ordered trace messages with minimal or no time information. Different processor timing characteristics would result in different trace port bandwidth requirements.

7. Conclusions

Growing complexity and sophistication of embedded computer platforms, a recent shift toward multicore architectures, and ever-tightening time-to-market make software development and debugging the most critical aspect of embedded system development. Improved on-chip tracing infrastructure coupled with sophisticated software debuggers promises to enable finding difficult and intermittent software bugs faster, resulting in higher quality software and increased overall productivity.

This paper introduces *mcfTRaptor*, a new low-cost technique for the on-the-fly capturing and compressing of control-flow traces in multicore systems. *mcfTRaptor* combines on-chip per-core private predictor structures and corresponding software counterparts in the software debugger to dramatically reduce the number of trace messages that need to be captured and streamed out on the target platform. The number of bits streamed out is further reduced by employing a variable encoding for counter and address fields in trace messages.

Our experimental evaluation explores the baseline trace port bandwidth requirements for control-flow traces in multicores and evaluates effectiveness of *mcfTRaptor* as a function of the number of cores, the configuration of predictor structures and their complexity, and the encoding mechanism. Our most effective configuration of *mcfTRaptor* reduces the trace port bandwidth between 23.8 times for functional control-flow traces and 18.6 times for time-stamped control-flow traces.

Acknowledgments

This material is based upon work supported in part by the United States National Science Foundation under Grants Nos. 0855237, 1205439, and 1217470. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. The authors would like to thank the anonymous reviewers for their valuable suggestions.

References

[1] G. Tassey, The economic impacts of inadequate infrastructure for software testing. [Online]. Available: http://www.rti.org/pubs/software_testing.pdf, 2002.
[2] IEEE-ISTO, The Nexus 5001 forum standard for a global embedded processor Debug Interface V 3.01, *Nexus 5001 Forum*, 2012. [Online]. Available: http://www.nexus5001.org/standard (accessed 27.09.14).
[3] IEEE, IEEE Standard 1149.1-2013 for Test Access Port and Boundary-Scan Architecture, *IEEE standards Association*, May-2013. [Online]. Available: http://standards.ieee.org/findstds/standard/1149.1-2013.html (accessed 27.09.14).
[4] W. Orme, Debug and trace for multicore SoCs, White paper, ARM, 2008.
[5] MIPS, MIPS PDtrace Specification Rev 7.50, MIPS Technologies Inc., CA, 2012.
[6] A. Mayer, H. Siebert, C. Lipsky, MCDS – multi-core debug solution, White paper, IPextreme, 2007.
[7] N. Stollon, R. Collins, Nexus Based Multi-Core Debug, in: Proceeding of the Design Conference International Engineering Consortium, Santa Clara, CA, USA, 2006, vol. 1, 805–822.
[8] C.-F. Kao, S.-M. Huang, I.-J. Huang, A hardware approach to real-time program trace compression for embedded processors, IEEE Trans. Circuits Syst. 54 (2007) 530–543.
[9] V. Uzelac, A. Milenkovic, A real-time program trace compressor utilizing double move-to-front method, in: Proceedings of the 46th Annual Design Automation Conference (DAC'09), July 26–31, San Francisco, CA, USA, 2009, 738–743.
[10] M. Milenkovic, A. Milenkovic, M. Burtscher, Algorithms and hardware structures for unobtrusive real-time compression of instruction and data address traces, in: Proceeding of the 2007 Data Compression Conference (DCC'07), Mar 27–29, Snowbird, UT, 2007, 55–65.
[11] A. Milenkovic, V. Uzelac, M. Milenkovic, M. Burtscher, Caches and predictors for real-time, unobtrusive, and cost-effective program tracing in embedded systems, IEEE Trans. Comput. 60 (7) (2011) 992–1005.
[12] V. Uzelac, A. Milenković, M. Burtscher, M. Milenković, Real-time unobtrusive program execution trace compression using branch predictor events, in: Proceeding of the International Conference on Compilers, Architectures and Synthesis of Embedded Systems (CASES'10), Scottsdale, AZ, 2010, 97–106.
[13] V. Uzelac, A. Milenković, M. Milenković, M. Burtscher, Using branch predictors and variable encoding for on-the-fly program tracing, IEEE Trans. Comput. 63 (4) (2014) 1008–1020.
[14] M. Williams, ARMV8 debug and trace architectures, in: Proceedings of the System, Software, SoC and Silicon Debug Conference (S4D), 2012, Vienna, 2012, 1–6.
[15] V. Uzelac, A. Milenković, Hardware-based data value and address trace filtering techniques, in: Proceeding of the International Conference on Compilers, Architectures and Synthesis for Embedded System (CASES'10), Scottsdale, AZ, USA, 2010, 117–126.
[16] V. Uzelac, A. Milenković, Hardware-based load value trace filtering for on-the-fly debugging, ACM Trans. Embed. Comput. Syst. 12 (2s) (2013) 1–18.
[17] C. Hochberger, A. Weiss, Acquiring an exhaustive, continuous and real-time trace from SoCs, in: Proceeding of the IEEE International Conference on Computer Design, ICCD 2008, 356–362.
[18] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, A. Gupta, The SPLASH-2 programs: characterization and methodological considerations, in: Proceeding of the 22nd Annual International Symposium on Computer Architecture, Santa Margherita Ligure, Italy, 1995, 24–36.

[19] SPLASH-2, 32 bit binary archive, *Multi2Sim benchmarks*. [Online]. Available: https://www.multi2sim.org/benchmarks/splash2.php (accessed 7.03.15).
[20] R. Ubal, B. Jang, P. Mistry, D. Schaa, D. Kaeli, Multi2Sim: a simulation framework for CPU–GPU computing, in: Proceedings of the 21st international conference on Parallel architectures and compilation techniques, Minneapolis, MN, USA, 2012, 335–344.
[21] K. Driesen, U. Hölze, Accurate indirect branch prediction, SIGARCH Comput. Arch. News 26 (1998) 167–178.
[22] S. McFarling, Combining Branch Predictors, Digital Equipment Corporation, 1993.
[23] V. Uzelac, A. Milenkovic, Experiment flows and microbenchmarks for reverse engineering of branch predictor structures, in: Proceeding of the IEEE International Symposium on Performance Analysis of Systems and Software, April 2009, Boston, MA, USA, 2009, 207–217.

Amrish K. Tewar received the B.Sc. degree in electrical engineering from the South Gujarat University, Surat, India in 2003. He is currently pursuing MS degree in computer engineering at University of Alabama in Huntsville (UAH) and expected to graduate in May 2015. His research interests include computer architecture, embedded systems, and mobile software development. He interned with Mentor Graphics Inc. during summer 2014. Prior to joining the UAH he worked in industry for eight years as an electrical engineer.

Albert Myers received the MS degree in computer engineering from the University of Alabama in Huntsville in 2014. His research interests include computer systems architecture. He is currently an engineer working in hardware design and failure analysis for the United States Army.

Aleksandar Milenković received the Dipl. Ing, MS, and PhD degrees in computer engineering and science from the University of Belgrade, Serbia, in 1994, 1997, and 1999, respectively. He is a professor of electrical and computer engineering at the University of Alabama in Huntsville, where he leads the LaCASA Laboratory (http://www.ece.uah.edu/~milenka). His research interests include computer architecture, embedded systems, VLSI, and wireless sensor networks. Prior to joining the University of Alabama in Huntsville, he held academic positions at the University of Belgrade in Serbia and the Dublin City University in Ireland. He has coauthored over 80 peer-reviewed research publications. He is a senior member of the IEEE, its Computer Society, the ACM, and Eta Kappa Nu.