Exploiting Cache Coherence for Effective On-the-Fly Data Tracing in Multicores

Mounika Ponugoti and Aleksandar Milenković
Department of Electrical and Computer Engineering
The University of Alabama Huntsville
Huntsville, AL, U.S.A
{mp0046, milenka}@uah.edu

***Abstract*—Software testing and debugging of modern embedded computer systems become increasingly a challenging task due to growing hardware and software complexity, increased integration and miniaturization, and ever tightening time-to-market. To find software bugs faster, developers often rely on on-chip trace and debug resources. However, these resources offer limited visibility of the system, increase the system cost, and do not scale well with a growing number of processor cores. This paper introduces a new hardware/software mechanism for capturing and filtering load data value traces in multicores that enables a complete reconstruction of a parallel program execution. The proposed mechanism exploits data caches and cache coherence protocol states to minimize the number of trace events that are necessary to stream out of the target platform to the software debugger. The mechanism relies on a single trace bit per data cache block, thus minimizing the cost of hardware implementation. Our experimental evaluation explores the effectiveness of the proposed technique by measuring the trace port bandwidth as a function of the cache size and the number of processor cores. The results show that the proposed mechanism significantly reduces the required trace port bandwidth when compared to the Nexus-like load data value tracing. Depending on data cache size, the improvements range from 9.9 to 23.5 times for single cores and from 18.6 to 37.3 times for octa cores.**

Keywords—Debugging aids, Tracing, Real-time embedded systems, Compression

I. INTRODUCTION

Increasing complexity and a shift to multicore architectures in modern embedded system make software development and testing critical aspects of system development. Faster and cheaper processors in smaller form factors enabled new applications that in turn increased users' expectations and their reliance on embedded systems. As a result, the complexity of the software stack in embedded systems keeps growing. A recent report from the International Technology Roadmap for Semiconductors found that the software engineering and tool costs account for 80% or more of the total development cost of modern high-end embedded systems [1]. Alas, the increasing software complexity has been accompanied by with tightening time-to-market. Software complexity and time-to-market pressures together lead to poorly tested software, lost revenue, or even project failures if time-to-market goals are not met.

It is important to give software developers tools to quickly locate and correct all software bugs with minimum effort. When debugging, software developers often need perfect visibility of the system state. However, achieving this visibility is not feasible due to high system complexity, limited available bandwidth for debugging data, and high operating frequencies. Traditional debugging techniques rely on single stepping, setting breakpoints, and examining the content of registers and memory locations while the processor is halted. This approach is effort- and time-consuming for software developers. In addition, it perturbs the sequence of events on target platforms and thus is not practical in real-time cyber-physical systems. Finally, it does not scale well to multicores.

To address these challenges, modern embedded processors increasingly rely on on-chip trace and debug infrastructure [2], [3], [4], [5]. Fig. 1 shows a block diagram of a system-on-a-chip (SoC) with N processor cores, a DSP, and a DMA core, all connected through a system interconnect. Each component includes its own tracing and debugging resources, called trace modules (see Fig. 1, excluding blue blocks). They are responsible for capturing and possibly filtering program execution traces and sending them to on-chip trace buffers through a debug interconnect. The program traces, temporarily stored in on-chip trace buffers, are streamed out of the chip through a dedicated trace port, typically to an external trace probe that interfaces a software debugger on a host workstation. These traces are then used by the software debugger to enable faithful program replay off-line. The IEEE Nexus 5001 standard [6] specifies four classes of debugging operations, including simple run-control debugging (Class 1), control-flow tracing (Class 2), data tracing (Class 3), and emulating memory and I/O through a trace port (Class 4). Each level progressively requires more on-chip resources and wider trace ports, thus increasing the system cost. The existing trace modules can capture full program execution traces for relatively small program segments only, due to limited capacity of on-chip trace buffers. Unfortunately, these traces are often insufficient to locate software bugs. With the growing complexity of the software running on embedded systems, the distance between the source of a bug and its manifestation may be in billions of instructions.

This paper focuses on data traces (Class 3 in Nexus 5001). They are critical in reconstructing program execution in multicores and uncovering bugs caused by data race conditions. In order to faithfully reconstruct a program execution in the software debugger, we need to capture data values of memory and I/O reads on the target platform and stream them out. In addition, we need to capture and report exceptions in the program flow. However, data value traces

tend to be very large, in the order of 8-16 bits per instruction executed per processor core [4]. Capturing data value traces in multicores is even more challenging because trace messages coming from different cores need to be ordered or time stamped before they are streamed out through a shared trace port. In addition, they need to include information about the origin of the trace message (core identification). Whereas a number of recent papers focuses on capturing, compressing, and filtering control-flow traces [7], [8], [9], [10], [11] relatively few studies look at on-the-fly data tracing [12]. Unfortunately, these studies exclusively focus on single-core embedded platforms where problem of ordering or time-stamping trace messages is not present. To the best of our knowledge there have been no academic studies focusing on hardware-supported data tracing in multicores.

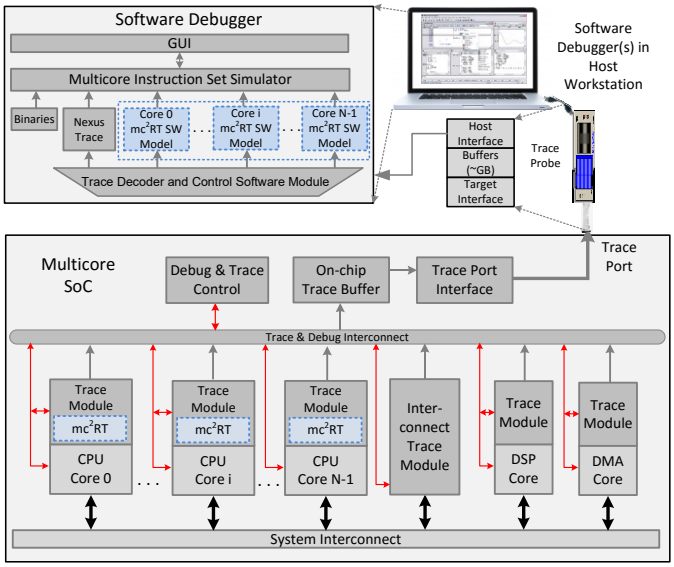

Fig. 1. Multicore debugging and tracing infrastructure

In this paper, we first analyze requirements for on-the-fly data tracing in multicores as a function of the number of cores by running a set of parallel programs (Section II). Next, we introduce mc²RT (multicore cache-coherent read trace), a hardware/software mechanism for capturing and filtering load data values in multicores. With mc²RT, each data cache block is associated with its trace bit that keeps track of whether the block has been traced out or not. mc²RT also relies on cache coherence protocols to ensure that actively shared cache blocks are traced out to the software debugger only once, the first time they are fetched from memory by a processor core (Section III). The mechanism relies on a sophisticated software debugger that maintains software copies of data caches and simulates their behavior during program replay. Our experimental evaluation (Section IV) explores the effectiveness of mc²RT as a function of the number of cores and data cache configurations. The results (Section V) indicate that mc²RT offers significant reduction in the required trace port bandwidth relative to the existing Nexus-like load data value tracing. Its effectiveness varies with the number of processor cores and the size of data cache. mc²RT reduces the trace port bandwidth from 9.9 times for N=1 to 18.6 times for N=8 when using 16 KB data caches, and from 23.5 times for N=1 to 37.3 times for N=8 when using 32 KB private data caches.

The main contributions of this work are as follows:

- We characterize trace port bandwidth requirements in multicores for Nexus-like time stamped and untimed memory read data value traces as a function of the number of cores. We consider both bits per instruction and bits per clock cycle as measures of the required trace port bandwidth.
- We develop a trace filtering technique called mc²RT for *multicore cache-coherent read trace* to reduce the trace port bandwidth requirements.
- We perform a detailed experimental evaluation of the trace port bandwidth, while varying the number of cores and cache sizes. In addition to analyzing the average trace port bandwidth per benchmark, we also consider variations of the trace port bandwidth during benchmarks' execution.

II. Data Tracing in Multicores

To faithfully replay a parallel program, a software debugger relies on the following artifacts: (a) an instruction set simulator for the target platform; (b) the binary of the parallel program; (c) the initial state of the target's general- and special-purpose registers; (d) exception traces; and (e) memory and input device read data value traces. The last two, the exception traces and the read data value traces, need to be captured on the target platform during the program execution and streamed out through the target's trace port. In multicores, both traces need to carry information about the inter- and intra-core ordering of trace events that are reported. Whereas intra-core ordering of trace events can be implemented using private trace buffers for each core, the inter-core ordering requires time-stamping trace events with a global time stamp. The time-stamped trace messages coming from different processor cores can be ordered in the global trace buffer and streamed out without time-stamps (referred to as untimed traces) or they are streamed unordered but with time stamps (referred to as timed traces). In this paper we consider both alternatives.

To illustrate the tracing challenges in multicores, we analyze the trace port bandwidth required by the read data value traces when running a suite of parallel programs on a multicore. The trace port bandwidth is reported in the average number of bits per instruction executed (bpi) and the average number of bits per processor clock cycle (bpc). The bandwidth in bpi is calculated as the total read data value trace size in bits divided by the number of instructions executed in a given benchmark. The bandwidth in bpc is calculated as the total read data value trace size in bits divided by the benchmark execution time measured in clock cycles. The trace port bandwidth depends on the number of instructions executed, the frequency of instructions that read data from memory, and data types. The bandwidth in bpc also depends on the multicore model (pipeline, out-of-order execution, caches, and other parameters), which can be characterized by the number of instructions committed in a clock cycle.

Fig. 2 shows SPLASH-2 benchmarks' characteristics of interest for data tracing [13][14]. The benchmarks are compiled for the Intel IA32 ISA and run on a cycle-accurate Multi2Sim simulator that models processors with N=1, 2, 4, and 8 cores. Fig. 2 graphs show (a) the number of instructions executed in

billions (IC), (b) the frequency of memory reads and (c) the number of instructions committed in each clock cycle (IPC). The number of instructions executed remains constant or slightly increases with an increase in the number of cores, with an exception of *cholesky* where the number of instructions increases significantly. The frequency of instructions reading data from memory increases slightly with an increase in the number of cores and varies from 13% for *fmm* to 35% for *radix*. The column Total shows the overall frequency for all benchmarks and is calculated as the sum of all memory reads divided by the sum of all instructions. The average IPC depends on the type of benchmarks, the target multicore model, and the number of cores. Thus, when N=1, the IPC ranges from 0.19 for *cholesky* to 0.66 for *water-sp*. The total IPC for the entire benchmark suite is calculated as the sum of all instructions executed by all benchmarks divided by the sum of all execution times in clock cycles. It ranges from 0.4 for N=1 to 1.95 for N=8. The IPC as a function of the number of cores indicates how well performance scales. Thus, *radix* scales poorly because its 8-core speedup is S(8)=IPC(8)/IPC(1)=2.8, but *water-ns* scales well because its 8-core speedup is S(8)=6.4.

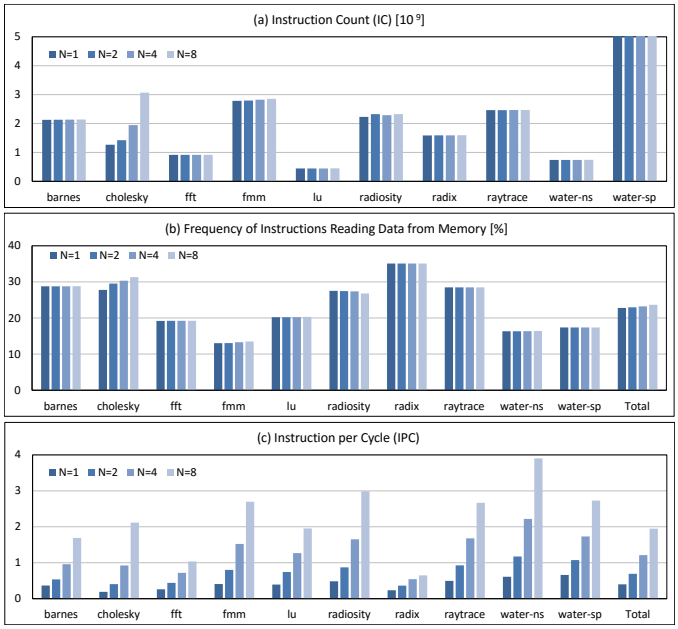

Fig. 2. Splash2 benchmark characterization

The Multi2Sim [15] simulator captures data values read from memory for committed instructions only. For untimed tracing, we assume that trace messages coming from individual cores contain time stamps. These time stamps are used by the global trace buffer control logic to order trace messages coming from different cores. The ordered trace messages are streamed out untimed, i.e., with no time stamp field. Each trace message includes a (Pi, LV) pair, where Pi represents the core index (equivalent to the thread index in our case) and LV represents the data value read from memory. We assume the software debugger can infer all other parameters (memory address, size of data) from the binary and the context maintained by the instruction set simulator(s). For time stamped trace messages, each trace message includes a (dCC, Pi, LV) triplet, where dCC represents the time in clock cycles measured from the beginning of the program execution or from the previous trace message at the given processor core.

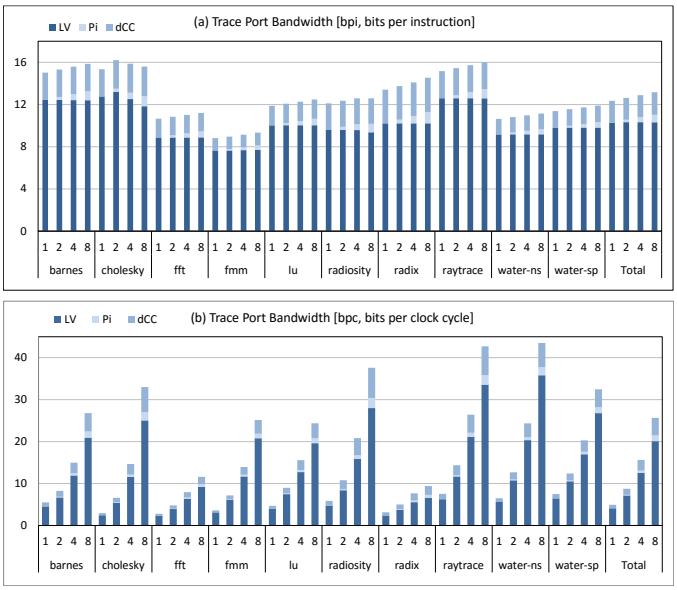

Fig. 3. Trace port bandwidth for Nexus-like load data value trace

Fig. 3a shows the trace port bandwidth (TPB) in bpi broken down into individual fields of trace messages. The TPB is highly correlated with the frequency of memory reads and the size of typical operands read from memory. For untimed traces the TPB ranges from 7.6 bpi for *fmm* to 12.8 bpi for *cholesky*, when N=1. It increases slightly with an increase in the number of cores due to (a) an increased overhead in reporting Pi and (b) an increase in the frequency of memory reads. When N=8, the TPB ranges from 8.1 bpi for *fmm* to 13.4 bpi for *raytrace*. The total trace port bandwidth for the entire benchmark suite is calculated as the sum of all trace messages in all benchmarks divided by the sum of all instructions executed in all benchmarks. It ranges from 10.3 bpi for N=1 to 11.0 bpi for N=8. Time-stamped trace messages include a differentially encoded time field, dCC (Fig. 8a). Consequently, the TPB increases relatively to untimed traces. The TPB ranges from 8.8 bpi for *fmm* to 15.4 bpi for *cholesky* when N=1, and from 9.3 bpi for *fmm* to 16 bpi for *raytrace* when N=8. The total trace port bandwidth for the time-stamped traces ranges from 12.3 bpi for N=1 to 13.2 bpi when N=8.

To further illustrate tracing challenges in multicores, we consider the trace port bandwidth in bpc (Fig. 3b). The required TPB for untimed read data value traces ranges from 2.3 for *fft* to 6.5 bpc for *water-sp* when N=1, and from 7.3 for *radix* to 37.7 bpc for *water-ns* when N=8. Benchmarks with a high frequency of memory reads that scale well with the number of cores (e.g., *raytrace*) place a lot of pressure on the trace port. The total trace port bandwidth for the entire benchmark suite ranges from 4.1 for N=1 to 21.5 bpc for N=8. In case of time-stamped read data value traces, the total TPB increases even further to 4.9 bpc when N=1 and to 25.6 bpc when N=8. For several benchmarks, such as *raytrace* and *water-ns*, the required TPB exceeds 42 bpc. It should be noted that the results in Fig. 3a and Fig. 3b indicate the average trace

port bandwidth for each benchmark. However, even higher peak rates at the trace port may occur during a benchmark execution. All these observations underscore a need for reducing the volume of trace data that needs to be streamed out of the chip.

III. MC²RT: MULTICORE CACHE-COHERENT READ TRACE

mc²RT is a hardware-based mechanism that filters the memory read data value traces by utilizing cache coherence protocols. Fig. 1 shows a block diagram of system debugging with dashed-line boxes representing additional mc²RT hardware and software modules. With mc²RT, each L1 data cache block in each processor core on the target platform is augmented with a trace tracking bit (Fig. 4). The trace bit keeps track of whether the associated cache block has been already traced out (T=1) or not (T=0). A cache block fetched from memory for the first time by a processor having a read miss will be traced out through the trace port. We refer to this event as a trace miss. Once a cache block is traced out, its corresponding trace bit is set (T=1). Previously traced cache blocks do not have to be traced out again as they can be inferred by the software debugger. We refer to cache read hits with the trace bit set as trace hits. This way, we can exploit the temporal and spatial locality of data accesses and cache coherence protocols to significantly reduce the number of trace messages. Each processor core also keeps a local trace-hit counter, THCnt, which counts the number of consecutive trace hits. In addition, a register keeps the time-stamp from the most recent trace event which is used to determine differentially encoded time-stamp for the next trace event.

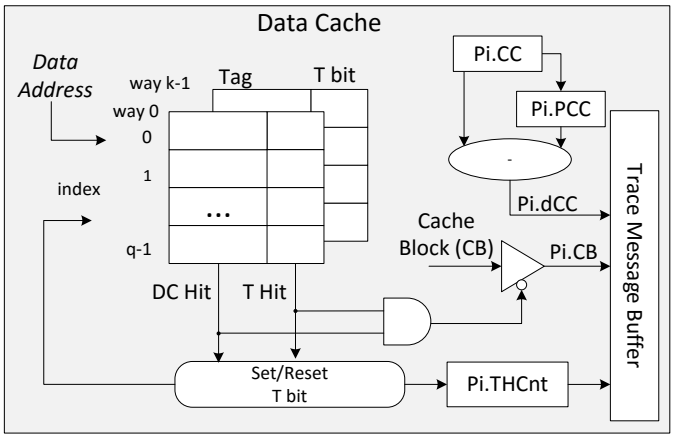

Fig. 4. mc²RT hardware structures for processor core *i*

Fig. 5 describes a sequence of events on a memory read by core *i*. The data cache lookup results in a cache hit or miss. In case of a cache hit and trace hit, the corresponding load data value does not need to be reported and the local THCnt counter is incremented (step 7). In case of a trace miss, a new trace message is emitted that includes a differentially encoded time-stamp (dCC=CC–PCC), processor core identifier (Pi), the current value of the trace counter (THCnt), and the content of the entire cache block (step 4). The trace hit counter is then cleared and the corresponding trace bit is set (step 5). In case of a cache miss, a coherent read transaction is issued (step 6). Without loss of generality we assume that the MOESI cache coherence protocol is used. The requested cache block is supplied to Pi either from another processor cache (Px) or from memory. If it is retrieved from another processor cache (Px), we assume that the trace bit is inherited by the processor Pi and a new state is *Shared* (step 11). If the block is retrieved from main memory, the corresponding T bit is cleared and a new cache block state is set to *Exclusive* (steps 12 and 13).

Fig. 6 describes a sequence of events on a memory write by core *i*. The data cache is looked up for the requested block. In case of a write hit in the *Exclusive* state (step 3), the state is upgraded to *Modified* (step 5). If the cache block is in the *Shared* or *Owned* state, a coherent invalidate transaction is initiated (step 4) to upgrade the cache block state to *Modified*. In case of a write miss, a Coherent Read and Invalidate transaction is initiated (step 6). The steps 7-12 describe important actions during this transaction. If the requested block is supplied by another processor cache, e.g. Px, the requesting processor T bit is inherited (step 10). If the cache block is retrieved from memory, the corresponding T bit is cleared (step 13).

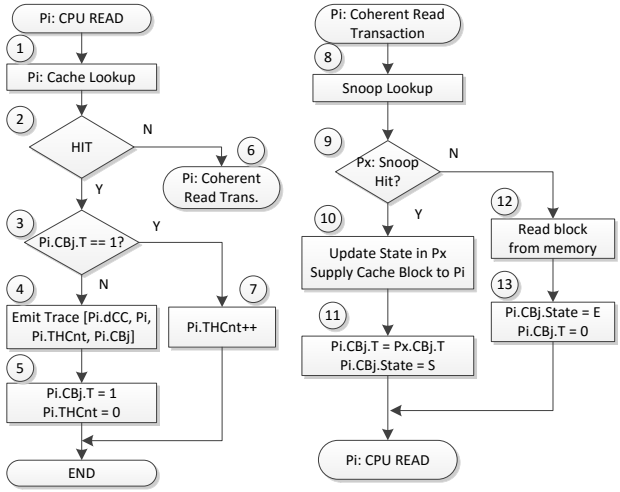

Fig. 5. mc²RT operation on the target processor core *i* for memory reads

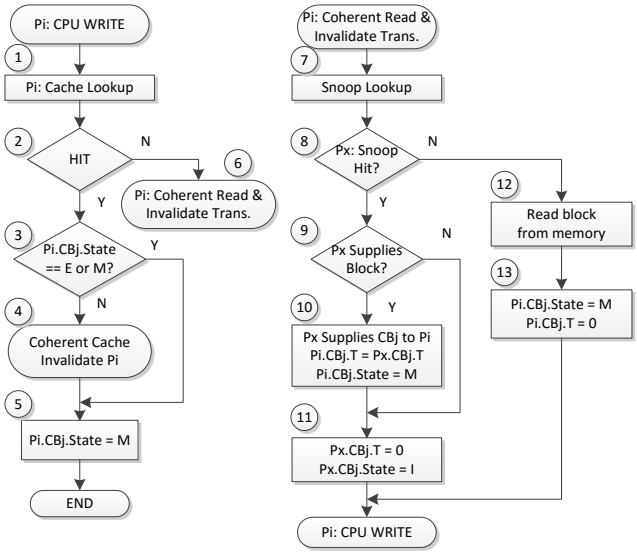

Fig. 6. mc²RT operation on the target processor core *i* for memory writes

The software debugger has access to the program binary, instruction set simulator, and trace messages streamed out from the target platform. The software debugger maintains software copies of data caches and trace hit counters. They are updated during the program replay using the same policies that are used on the target platform. The debugger replays the instructions for each processor core using instruction set simulator(s). For each memory read instruction (Fig. 7), the corresponding Pi.THCnt is decremented (the initial value in the software debugger is set to 1). If Pi.THCnt>0, the debugger retrieves the operand from the software copy of the data cache (either from the private or a remote) and moves to replay the next instruction. If Pi.THCnt=0, we have a trace miss event. The cache block portion of the trace message is extracted and used to update the corresponding cache block in the software data cache for processor Pi. Then, the next trace message is retrieved from the trace buffer and decoded. The software copy of the Pi.THCnt is loaded with a value extracted from the trace message. For each memory write operation, the software copy of the data cache is updated following the same steps as on the target platform (Fig. 7).

mc^{2}RT requires relatively minor hardware extensions to support data tracing. The majority of hardware overhead is due to the trace bits. If we assume processor cores with a 32 KB data cache and 32 B cache blocks, the overhead is 1,024 bits or 128 B of additional storage in the cache. It should be noted that mc^{2}RT on a trace miss emits the entire cache block, not just requested load data value. One can argue that in cases of poor spatial locality certain portions of the cache block will not be needed, yet their streaming out will consume trace port bandwidth. An alternative approach is to use multiple trace bits per cache as described in [12]. However, that approach will prevent or make challenging to utilize cache coherence protocols in reducing the number of trace messages emitted.

```
1.  // For each memory read on processor core i, Pi
1. Pi.THCnt--;
2. if (Pi.THCnt > 0) {
3.     Perform lookup in the SW data cache;
4.     Retrieve data value from SW data cache;
5. }
6. else { // T miss event
7.     Read cache block from the trace record;
8.     Update SW cache;
9.     Get next trace message (Pi.dCC, Pi, Pi.THCnt, Pi.CB);
10.}

11.// For each memory write on Pi
12.Update SW cache;
```

Fig. 7. mc^{2}RT operations in the software debugger on processor core *i*

A. Encoding of trace messages

Trace messages streamed out through the trace port should be encoded in such a way to minimize the number of bits. Fig. 8 shows formats of trace messages for the Nexus-like load value trace (NX) and for mc^{2}RT. With NX, each load data value (*LV*) is streamed out through the trace port together with core index on which the read operation is carried out (*Pi*) and differentially encoded time stamp (*dCC*). The length of the *Pi* field is fixed and is a function of the number of cores (0 bits for N=1, 1 bit for N=2, 2 bits for N=4). In NX, the length of the *LV* field depends on the size of the operand read from memory (for IA32 ISA it ranges from 1 to 120 bytes) and is thus *8·sizeof(type)* bits. The time field, *dCC*, carries information about the clock cycle in which the current trace-generating instruction has retired. Rather than recording the absolute clock cycle from the beginning of the program, it contains the number of clock cycles expired from the previous trace event on the core *i*, *dCC = CC–PCC*. Note: the first trace message contains the time from the beginning of the program. For simplicity, we assume all cores share a global clock. The number of bits needed to encode *dCC* varies among programs and during program execution. With NX and mc^{2}RT we use at least 8 bits to encode *dCC*. The connect bit (C) determines whether more 8-bit chunks are needed to fully encode *dCC* value (C=1) or not (C=0).

mc^{2}RT trace messages consist of the following fields: *dCC*, *Pi*, *THCnt*, and *CB*. The *dCC* and *Pi* fields are encoded in the same way as in NX. The *THCnt* field contains the value of the *THCnt* counter. The number of bits needed to encode *THCnt* varies as a function of trace miss rate. We use at least 8 bits to encode this field. The connect bit (C) determines whether more 8-bit chunks are needed to fully encode *THCnt* value (C=1) or not (C=0). The length of the CB field corresponds to the cache block size. For example, if the cache block size is 32 bytes, the size of the CB field is 256 bits.

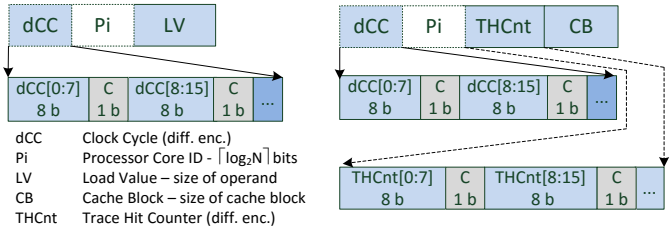

Fig. 8. Formats of trace messages

IV. Experimental Environment

The goal of the experimental evaluation is to determine the effectiveness of the proposed mc^{2}RT as a function of the number of cores and cache configuration. As a measure of effectiveness, we use the average trace port bandwidth requirements expressed in bpi and bpc. Whereas the average TPB allows us to quantify the effectiveness of the proposed technique, it does not fully capture the peak rates that occur in individual benchmarks during their execution. Consequently, we also analyze the TPB as a function of time during benchmark execution.

Fig. 9 shows the experimental flow used to create hardware traces and evaluate the trace port bandwidth. The timed traces are collected using the Multi2Sim [15] simulator executing IA32 ISA binaries. The simulator is extended with a custom TmTrace module that captures time-stamped memory read and write traces (tmlsTrace). The time stamp contains the global clock cycle in which the trace-generating instruction is committed. The tmlsTraces are read, filtered, and encoded to generate the Nexus-like trace, NX. The tmlsTraces are also read by the mc^{2}RT simulator that generates filtered memory read data value traces. The output traces are then processed by encoding tools that determine trace port bandwidth and generate minimal hardware traces, namely mc^{2}RT. As the workload we use Splash2 benchmarks run with N=1, 2, 4, and 8 cores.

The Multi2Sim simulator supports building a cycle-accurate model for a multicore processor including processor and memory hierarchy. We use a multicore with up to 8 single-threaded x86 processor cores as shown in Fig. 10. Each core has its private level 1 instruction (L1I) and data (L1D) caches with hit latency of 4 clock cycles. To evaluate effectiveness of mc^{2}RT as a function of the cache size, we consider two configurations of caches: CS16 with 16 KB L1D, and CS32 with 32 KB L1D. The L1 data caches are 4-way set-associative with LRU replacement policy, and 32 byte cache blocks. The unified L2 cache memory is shared by all cores and has a hit latency of 12 clock cycles. The L2 cache size varies with the number of cores, N, and it is set to N·64KB for the CS16 configuration and N·128KB for the CS32 configuration. The main memory latency is set to 100 clock cycles.

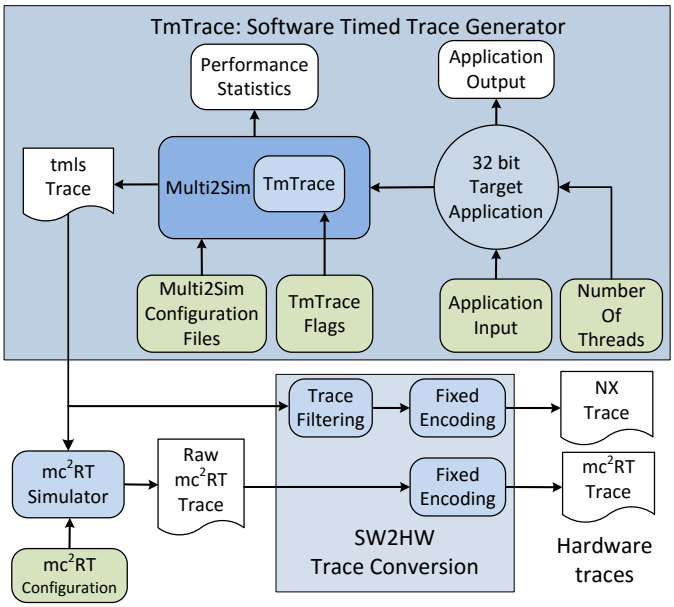

Fig. 9. Experimental environment

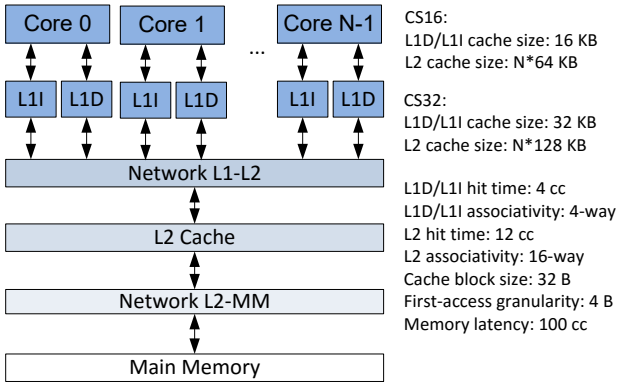

Fig. 10. Multicore model

V. Results

The effectiveness of mc^{2}RT directly depends on (a) benchmark characteristics – namely, the type, frequency, and distribution of memory read operations, (b) data cache miss rates and trace-bit miss rates, and (c) encoding parameters. The trace miss rate is a good indicator of mc^{2}RT effectiveness – the lower it is, the fewer trace messages need to be streamed out through the trace port. The trace miss rate is very close to the cache read miss rate though not identical, because cache blocks can be brought to the cache by write misses too. In addition, with an increase in the number of processor cores, the portion of truly shared data is growing, and thanks to our mechanism to inherit tracing bits during cache coherent transactions, we will avoid tracing cache blocks that have been previously reported by other processors. Fig. 11 shows the total read L1 data cache miss rate and the total trace miss rate for the entire benchmark suite as a function of the number of cores and the data cache configurations (CS16 and CS32). It also shows the minimum and the maximum miss rates. The total L1 data cache read miss rate is calculated as the total number of read misses divided by the total number of read requests when all benchmarks are considered together. The total trace miss rate is calculated as the total number of trace misses divided by the total number of data reads when all benchmarks are considered together. For the CS16 configuration, the read L1 data cache miss rate is below 1.7% regardless of the number of cores, with maximum rate below 4.7%. For the CS32 configuration, the read L1 data cache miss rate is below 0.7% with the maximum ~2.7%. The trace miss rate decreases with an increase in the number of cores, from 1.9% when N=1 to 1.0% when N=8 for the CS16 configuration and from 0.8% when N=1 to 0.5% when N=8 for the CS32 configuration. The maximum trace miss rate reaches as high as 4.7% with the CS16 configuration and 2.7% for the CS32 configuration (*fft* benchmark). Overall, the results confirm our expectations that mc^{2}RT can indeed significantly reduce the number of trace messages that needs to be streamed out through the trace port.

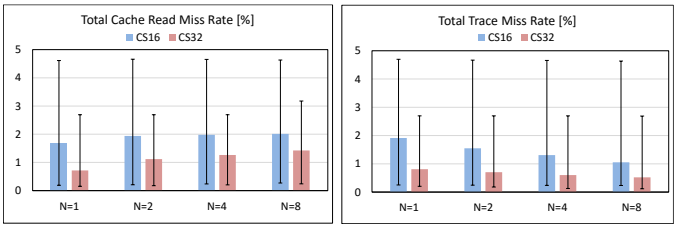

Fig. 11 Data cache read miss rate & trace miss rate

A. Trace port bandwidth in bpi

Fig. 12 shows the total average trace port bandwidth with the min-max ranges in bpi for the timed Nexus-like data value traces (NX) and the timed mc^{2}RT traces as a function of the number of cores and cache configurations. NX requires from 12.3 bpi when N=1 (ranging from 8.8 bpi for *fmm* to 15.3 bpi for *cholesky*) to 13.2 bpi when N=8 (ranging from 9.3 bpi for *fmm* to 16 bpi for *raytrace*). mc^{2}RT dramatically reduces the total trace port bandwidth requirements relative to NX in CS16 configuration. It requires from 1.24 bpi when N=1 (ranging from 0.12 for *water-sp* to 4.03 for *barnes*) to 0.71 bpi when N=8 (from 0.12 for *water-sp* to 2.49 for *fft*). Compared to NX, mc^{2}RT thus reduces the bandwidth 9.9 times for N=1 and 18.6 times for N=8 (see Table 1). Expectedly, increasing the data caches leads to even lower trace port bandwidths. Thus, mc^{2}RT with CS32 requires only 0.52 bpi regardless of the number of cores (ranging from 0.1 for *water-sp* to 1.45 bpi for *fft*). Compared to NX, mc^{2}RT(CS32) reduces the trace port bandwidth 23.5 times when N=1 and 37.3 times when N=8.

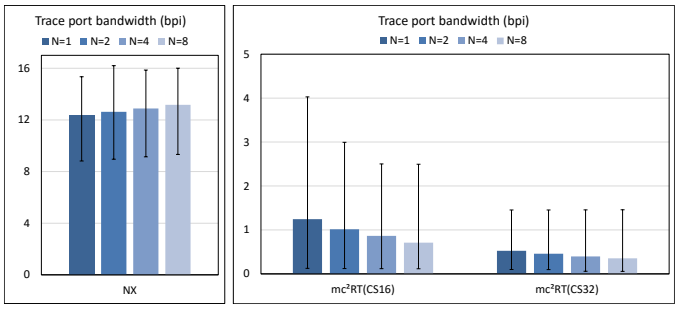

Fig. 12 Trace port bandwidth in bpi

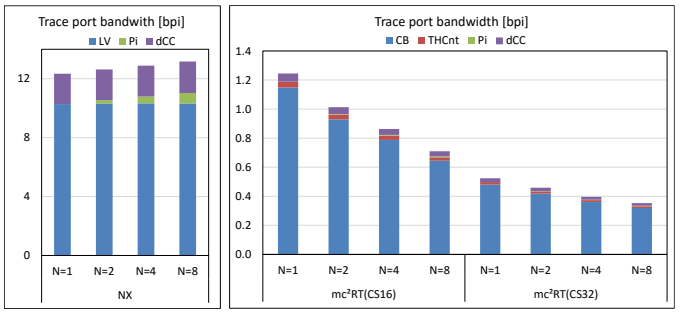

Fig. 13 Trace port bandwidth of indivudal trace fields in bpi

TABLE I. SPEEDUPS ACHIEVED BY GZIP AND MC²RT

# Cores	N=1			N=2			N=4			N=8		
Config	NX.gz	CS16	CS32	NX.gz	CS16	CS32	NX.gz	CS16	CS32	NX.gz	CS16	CS32
barnes	1.4	3.7	10.7	1.3	5.1	14.6	1.2	7.2	19.6	1.3	9.9	23.7
cholesky	1.7	8.7	22.1	1.5	13.2	22.4	1.2	18.4	29.8	1.0	29.9	43.8
fft	1.4	4.3	7.3	1.4	4.3	7.5	1.3	4.4	7.6	1.4	4.5	7.7
fmm	1.9	16.2	23.9	1.8	18.0	25.1	1.6	19.1	26.0	1.6	19.9	27.4
lu	1.6	23.8	24.1	1.5	24.2	26.2	1.4	43.5	49.2	1.8	46.8	73.5
radiosity	1.6	22.2	62.2	1.5	40.0	88.9	1.3	43.2	101.0	1.4	59.7	124.5
radix	2.0	9.4	19.7	1.8	9.6	20.0	1.5	9.8	20.3	1.4	10.0	20.6
raytrace	1.5	7.7	24.7	1.5	9.8	33.4	1.3	11.1	38.2	1.4	12.9	45.4
water-ns	1.4	12.9	28.4	1.4	13.1	28.6	1.3	17.3	185.1	1.3	74.7	200.2
water-sp	1.4	91.5	114.2	1.4	95.5	119.0	1.3	99.8	128.4	1.4	103.1	138.7
Total	1.5	9.9	23.5	1.5	12.5	27.5	1.3	14.9	32.5	1.4	18.6	37.3

To underscore effectiveness of mc²RT, we compare its trace port bandwidth to the trace port bandwidth we can achieve by using a general-purpose compressor. The NX trace is used as an input to the software gzip utility with compression level 1. Table 1 shows the compression ratio achieved by the gzip utility (columns marked as NX.gz). The results show a limited total compression ratio of 1.5 for N=1 (ranging from 1.4 to 2) and 1.4 for N=8 (ranging from 1 to 1.8). The results confirm that redundancy of input load values is fairly limited and that general-purpose compressors would not be effective in reducing load data value traces. In addition, implementing them in hardware would impose significant complexity because of buffering and computation modules.

Fig. 13 shows the total trace port bandwidth in bpi broken down into individual fields of trace records: *Pi*, *THCnt*, *dCC*, and data values (*LV/CB*). Expectedly, the majority of trace port bandwidth is consumed by streaming out data values. In NX, the *LV* portion ranges from 83% for N=1 to 78% for N=8. The time field is responsible for ~16% of the bandwidth regardless of the number of processors. Thus, if we order trace records from different cores in the trace buffer and stream them out without a time field, the trace port bandwidth requirements will be lower. In mc²RT, the data field (CB in the trace message) accounts for the majority of the trace port bandwidth.

B. *Trace port bandwidth in bpc*

Fig. 14 shows the total trace port bandwidth with the min-max ranges in bpc for the timed NX and mc²RT traces. The total trace port bandwidth for NX scales linearly with the number of cores, from 4.9 bpc for N=1 (from 2.8 to 7.5 bpc) to 25.6 bpc for N=8 (from 9.4 to 43.5 bpc). mc²RT provides significant reductions in the trace port bandwidth. Thus, mc²RT(CS16) requires from 0.5 bpc for N=1 (from 0.08 to 1.47 bpc) to 1.38 bpc for N=8 (from 0.32 to 3.3 bpc). mc²RT(CS32) requires from 0.23 bpc (from 0.07 to 0.58) for N=1 to 0.70 bpc (from 0.22 to 1.54) for N=8.

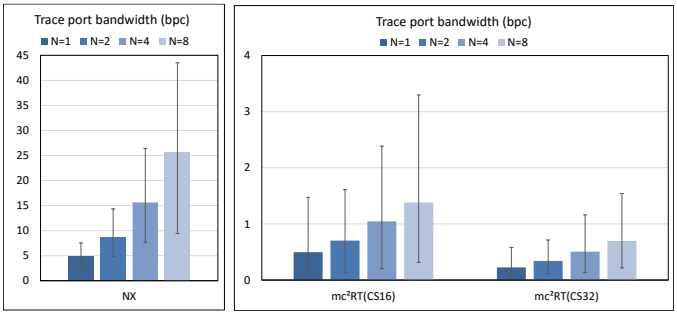

Fig. 14 Trace port bandwidth in bpc

C. *Dynamic Trace Port Bandwidth Analysis*

Whereas the average trace port bandwidth allows us to quantify the effectiveness of mc²RT, it does not fully capture the peak rates that occur in individual benchmarks during their execution. Depending on frequency and distribution of memory reads and trace misses, the trace port bandwidth at a given moment in a program execution may exceed the average bandwidth discussed above.

Fig. 15 and Fig. 16 show the trace port bandwidth during execution of two benchmarks, *raytrace* and *water-ns*, respectively. The number of cores is set to N=8. We analyze the bandwidth required for time-stamped NX and mc²RT traces with both configurations, CS16 and CS32. The benchmarks *raytrace* and *water-ns* are selected because they require the highest average total bandwidth for time-stamped load data value traces. The trace port bandwidth in bpc is logged every 1 million clock cycles.

Let us first analyze the bandwidth as a function of time for *raytrace*. The average trace port bandwidth is 42.7 bpc for NX(CS16) and 45.8 bpc for NX(CS32). However, the peak bandwidth reaches ~61 bpc, further underscoring the challenges in program tracing. mc²RT(CS16) requires the average bandwidth of 3.3 bpc with peak values of 6.6 bpc, an order of magnitude smaller bandwidth than for NX. mc²RT(CS32) requires the average bandwidth of 1.0 bpc with the peak value of 5.4 bpc. These results indicate that the mc²RT not only reduces the average trace port bandwidth, but also reduces the requirements for on-chip trace buffers. Similar observations stand for *water-ns*. The average trace port bandwidth is 43.5 bpi for NX(CS16) and 43.3 for NX(CS32) with the peak bandwidth of 56.4 bpc. mc²RT(CS16) requires

~0.6 bpc with the peak of 2.8 bpc. mc²RT(CS32) requires ~0.2 bpc with the peak of 2.4 bpc.

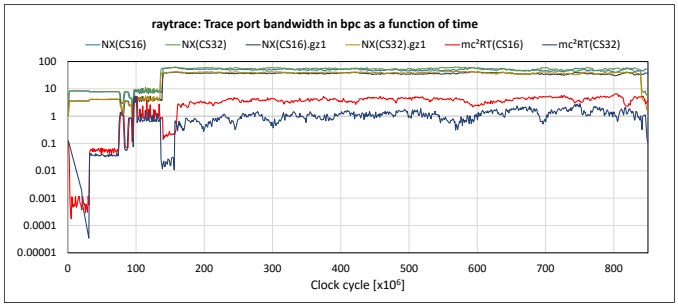

Fig. 15 Dynamic trace port bandwidth in bpc during execution of *raytrace* for N=8

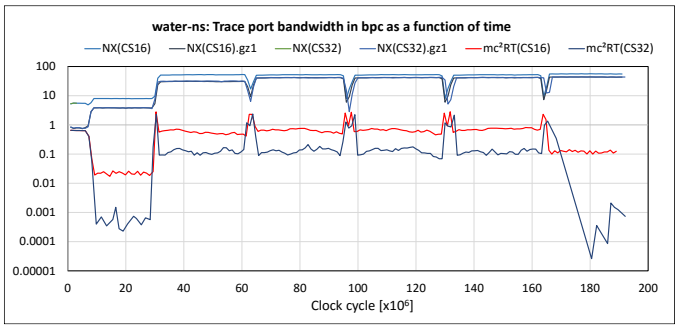

Fig. 16 Dynamic trace port bandwidth in bpc during execution of *water-ns* for N=8

VI. Conclusions

Growing complexity of hardware and software stacks, a recent shift toward multicores, and ever-tightening time-to-market make software testing and debugging one of the most critical aspects of embedded system development. Improved on-chip debugging and tracing infrastructure, coupled with sophisticated software debuggers, promises to reduce time and effort in finding difficult and intermittent bugs, thus resulting in higher quality software and increased productivity.

This paper introduces mc²RT, a technique for on-the-fly capturing and filtering load data value traces in multicore systems. mc²RT requires minimal extensions of data caches to include trace tracking bits, as well as software copies of data caches maintained by the software debugger. The trace tracking bits, updated by memory read and write operations, determine which memory read operations need to be streamed out to the software debugger. By exploiting cache coherence protocol states, mc²RT minimizes chances that a single cache block is reported multiple times.

Our simulation-based experimental evaluation explores the effectiveness of mc²RT as a function of data cache sizes (16 and 32 KB) and the number of processor cores (N=1, 2, 4, and 8). As a measure of the effectiveness, we use the trace port bandwidth expressed in the number of bits streamed on the trace port per instruction executed and the number of bits per processor clock cycle. mc²RT compression ratio relative to the Nexus-like load data value traces ranges: from 9.9 (N=1) to 18.6 (N=8) times for the configuration with 16 KB data caches; and from 23.5 (N=1) to 37.3 (N=8) times for the configuration with 32 KB data caches.

Acknowledgment

This work was supported in part by US National Science Foundation (NSF) grant CNS-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.

References

[1] "International Technology Roadmap for Semiconductors 2007 Edition." [Online]. Available: https://goo.gl/TdZY52. [Accessed: 08-Apr-2016].

[2] "MCDS - Infineon Multi-Core Debug Solution." [Online]. Available: https://www.ip-extreme.com/IP/mcds.shtml. [Accessed: 01-Apr-2016].

[3] MIPS, "MIPS PDtrace Specification," 2009. [Online]. Available: http://goo.gl/UwIYGv.

[4] W. Orme, "Debug and Trace for Multicore SoCs," 2008. [Online]. Available: http://goo.gl/Wrc7Hk. [Accessed: 28-Mar-2016].

[5] N. Stollon and R. Collins, "Nexus Based Multi-Core Debug," in *Proceedings of the Design Conference International Engineering Consortium*, Santa Clara, CA, USA, 2006, vol. 1, pp. 805–822.

[6] IEEE-ISTO, "The Nexus 5001 Forum Standard for a Global Embedded Processor Debug Interface," 2003. [Online]. Available: http://goo.gl/RZPYXU. [Accessed: 28-Mar-2016].

[7] C.-F. Kao, S.-M. Huang, and I.-J. Huang, "A Hardware Approach to Real-Time Program Trace Compression for Embedded Processors," *IEEE Trans. Circuits Syst.*, vol. 54, no. 3, pp. 530–543, Mar. 2007.

[8] B. Mihajlović and Ž. Žilić, "Real-time Address Trace Compression for Emulated and Real System-on-chip Processor Core Debugging," in *Proceedings of the 21st edition of the great lakes symposium on Great lakes symposium on VLSI*, New York, NY, USA, 2011, pp. 331–336.

[9] A. Milenković, V. Uzelac, M. Milenković, and B. Burtscher, "Caches and Predictors for Real-Time, Unobtrusive, and Cost-Effective Program Tracing in Embedded Systems," *IEEE Trans. Comput.*, vol. 60, no. 7, pp. 992–1005, Jul. 2011.

[10] V. Uzelac, A. Milenković, M. Milenković, and M. Burtscher, "Using Branch Predictors and Variable Encoding for On-the-Fly Program Tracing," *IEEE Trans. Comput.*, vol. 63, no. 4, pp. 1008–1020, Apr. 2014.

[11] V. Uzelac and A. Milenkovic, "A Real-Time Program Trace Compressor Utilizing Double Move-to-Front Method," in *Proceedings of the Design Automation Conference*, San Francisco, CA, 2009, pp. 738–743.

[12] V. Uzelac and A. Milenkovic, "Hardware-Based Load Value Trace Filtering for On-the-Fly Debugging," *Trans. Embed. Comput. Syst.*, vol. 12, no. 2s, pp. 1–18, May 2013.

[13] "Multi2Sim/m2s-bench-splash2," *GitHub*. [Online]. Available: https://goo.gl/5kbE8r. [Accessed: 01-Apr-2016].

[14] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, 1995, pp. 24–36.

[15] R. Ubal, B. Jang, P. Mistry, D. Schaa, and 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, pp. 335–344.